CS 161 (Stanford, Winter 2022) Lecture 1

Adapted From Virginia Williams' lecture notes. Additional credits: J. Su, W. Yang, Gregory Valiant, Mary Wootters, Aviad Rubinstein.
Please direct all typos and mistakes to Nima Anari and Moses Charikar.

Introduction

1 Logistics

The class website is at `https://cs161.stanford.edu`. All course information is available on the website.

2 Why are you here?

Many of you are here because the class is required. But why is it required?

1. **Algorithms are fundamental to all areas of CS:** Algorithms are the backbone of computer science. Wherever computer science reaches, an algorithm is there, and the classical algorithms algorithmic design paradigms that we cover re-occur throughout all areas of CS. For example, CS 140 and 143 (operating systems and compiles) leverages scheduling algorithms and efficient data structures, CS 144 (networking) crucially uses shortest-path algorithms, CS 229 (machine learning) leverages fast geometric algorithms and similarity search, CS 255 (cryptography) leverages fast number theoretic and algebraic algorithms, and CS 262 (computational biology) leverages algorithms that operate on strings–and often employs the dynamic programming paradigm. We'll discuss applications to all these subjects in this class.

   Algorithms and the computational perspective (the "computational lens") has also been fruitfully applied to other areas, such as physics (e.g. quantum computing), economics (e.g. algorithmic game theory), and biology (e.g. for studying evolutions, as a surprisingly efficient algorithm that searches the space of genotypes).

2. **Algorithms are useful:** Much of the progress that has occurred in tech/industry is due to the dual developments of improved hardware (a la "Moore's Law"—a prediction made in 1965 by the co-founder of Intel that the density of transistors on integrated circuits would double every year or two), and improved algorithms. In fact, the faster computers get, the bigger the discrepancy is between what can be accomplished with fast algorithms vs what can be accomplished with slow algorithms. . . . Industry needs to continue developing new algorithms for the problems of tomorrow, and you can help contribute.

3. **Algorithms are fun!** The design and analysis of algorithms requires a combination of

creativity and mathematical precision. It is both an art and a science, and hopefully at least some of you will come to love this combination. One other reason it is so much fun is that algorithmic surprises abound. Hopefully this class will make you re-think what you thought was algorithmically possible, and cause you to constantly ask “is there a better algorithm for this task?”. Part of the fun is that Algorithms is still a young area, and there are still many mysteries, and many problems for which (we suspect that) we still do not know the best algorithms. This is what makes research in Algorithms so fun and exciting, and hopefully some of you will decide to continue in this direction.

3 Karatsuba Integer Multiplication

3.1 The problem

Suppose you have two large numbers, and you want to multiply them. Of course, you all know how to solve this problem: you learned an algorithm (which we’ll call the “grade-school algorithm") when you were in grade school. The question is, **can we do better?**

In order to understand this, we need to talk at least a little bit about what we mean by better. How do we measure the running time of an algorithm? It’s tempting to measure it in units of time—say, milliseconds on a computer. However, this doesn’t really capture the running time of an *algorithm.* Rather, it captures the running time of an algorithm, with a particular implementation, on a particular piece of hardware. For example, grade-school multiplication is much faster on a computer than by hand, but it’s still the same algorithm in both cases.

Grade school algorithm

Instead, we’ll focus on how fast the running time *scales* as a function of the input. We will be a bit more precise about this in the next lecture, but for now, we’ll define this notion by example. Suppose we use the grade-school algorithm to multiply two n-digit numbers. The bulk of the work is taken up by multiplying every pair of digits together. For example, in 1234×6789, we have to multiply $9 \times 4, 9 \times 3, 9 \times 2, 9 \times 1, 8 \times 3$, etc. There are n^2 such pairs, so we’ll say that this algorithm has a running time that scales like n^2.

Why should we care about this measure of complexity? We’ll talk about this more next lecture, but intuitively, this scaling behavior is the thing that really matters as n gets large. Suppose we had two algorithms, one of which had running time that scaled like n^2 and one which scaled like $n^{1.6}$. Suppose that running an algorithm by hand is 10000 times slower than running an algorithm on a computer. For large enough n, $10000n^{1.6} < n^2$, and intuitively this means it would actually be faster to run the $n^{1.6}$ algorithm by hand than the n^2 algorithm on a computer. So we can definitively say that the $n^{1.6}$ algorithm is “faster," because for large n, it will be faster, no matter how the algorithm is implemented and no matter what hardware it’s running on.

With that in mind, our question is now this: can we multiply two n-digit integers faster than the grade-school algorithm? That is, with a running time that scales faster than n^2?

One try might be to store the answers ahead of time, or at least store partial answers. For example, we could store the products of all pairs of n-digit numbers, and then just look up the pair we need. This does result in performance gains, however, and also leads to exponential storage costs. (For example, if $n = 100$, we would need to store a table of $10^{2n} = 10^{200}$ products. Note that the number of atoms in the universe is only $\approx 10^{80}$. . . .) So we'll have to do something more clever.

3.2 Divide and Conquer

The "Divide and Conquer" algorithm design paradigm is a very useful and widely applicable technique. We will see a variety of problems to which it can be fruitfully applied. The high-level idea is just to split a given problem up into smaller pieces, and the solve the smaller pieces, often recursively.

How can we apply Divide and Conquer to integer multiplication? Lets try splitting up the numbers. For example, if we were multiplying 1234×5678, we could express this as $((12 \cdot 100) + 34) \cdot ((56 \cdot 100) + 78)$. In general, if we are multiplying two n-digit numbers x and y, we can write $x = 10^{n/2} \cdot a + b$ and $y = 10^{n/2} \cdot c + d$. So

$$x \cdot y = (10^{n/2}a + b) \cdot (10^{n/2}c + d) = 10^n ac + 10^{n/2}(ad + bc) + bd.$$

Now we can split this problem into four subproblems, where each subproblem is similar to the original problem, but with half the digits. This gives rise to a recursive algorithm.

Interestingly enough, **this algorithm isn't actually better!** Intuitively this is because if we expand the recursion, we still have to multiply every pair of digits, just like we did before. But in order to prove this formally, we need to formally define the runtime of an algorithm, and prove that these algorithms are not very different in runtime.

3.3 Recurrence Relation

We can analyze the runtime of the algorithm as follows.

Let $T(n)$ be the runtime of the algorithm, given an input of size n (two n-digit numbers). Because we are breaking up the problem into four subproblems with half the digits, plus some addition with linear cost, we have the equation $T(n) = 4T(n/2) + O(n)$. (Don't worry if you haven't seen big-O notation before; we'll go over this in detail in the coming lectures.)

Although **in general you should pay attention to the $O(n)$ term**, today we will just ignore it because the term doesn't matter in this case.

By repeatedly breaking up the problem into subproblems, we find that

$$T(n) = 4T(n/2) = 16T(n/4) = \ldots = 2^{2t}T\left(\frac{n}{2^t}\right) = n^2 T(1).$$

Since $T(1)$ is the time it takes to multiply two digits, we see that the above suggestion does not reduce the number of 1-digit operations.

Note: In the lecture slides, we'll consider a slightly different argument, which analyzes a recursion tree. It's a good exercise to understand both arguments! Again, we'll discuss both techniques more in coming lectures.

3.4 Divide and conquer (take 2)

Karatsuba found a better algorithm (in 1960, published in 1962) by noticing that we only need the sum of ad and bc, not their actual values. So he improved the algorithm by computing ac and bd as before, and computing $(a+b)\cdot(c+d)$. It turns out that if $t=(a+b)\cdot(c+d)$, then $ad+bc=t-ac-bd$. Now instead of solving four subproblems, we only need to solve three! This idea goes back to Gauss, who found a similar efficient way to multiply two complex numbers.

Sure, we need to do more additions, but again it turns out that additions are pretty cheap. To do a quick-and-dirty analysis of the number of operations required by Karatsuba multiplication, first assume that $n=2^s$ for some integer s. (Note that we can always add 0's to the front of a number until the length is a power of two, so this assumption holds without loss of generality.) Letting $T(n)$ denote the number of multiplications of pairs of 1-digit numbers required to compute the product of two n-digit numbers, Karatsuba's algorithm gives $T(n)=3T(n/2)$, since we've divided the problem into three recursive calls to multiplication of length $n/2$ numbers. [Note, we are cheating a bit here, since $(a+b)$ and $(c+d)$ might actually be $n/2+1$ digit numbers, but lets ignore this for now...] Hence we have the following:

$$T(n)=3T(n/2)=3^2T(n/4)=\ldots=3^sT(n/2^s).$$

Since we assumed $n=2^s$, we have that $T(n/2^s)=T(1)=1$, since multiplying two 1-digit numbers counts as 1 basic operation. Hence $T(n)=3^s$, where $n=2^s$. Solving for s yields $s=\log_2 n$, and hence we get

$$T(n)=3^{\log_2 n}=2^{(\log_2 3)(\log_2 n)}=n^{\log_2 3}\leq n^{1.6}.$$

We were pretty sloppy with the above argumentation in a lot of ways. However, we'll see a much more principled way of analyzing the runtime of recursive algorithms in the coming classes, so we won't sweat about it too much now. The point is that (even if you do it correctly) the running time of this algorithm scales like $n^{1.6}$. Thus is **much** better than the n^2 algorithm that we learned in grade school!

3.5 Can we do better?

Progress on efficient algorithms for multiplication of n-digit numbers continued beyond Karatsuba's algorithm. Although you don't need to know these algorithms for CS 161, it is interesting to review the history of progress on this problem. Toom and Cook (1963) developed an algorithm that ran in time $O(n^{1.465})$ by showing how a single n-sized problem could be broken up into five $n/3$-sized problems. Schönage and Strassen (1971) developed an algorithm that runs in time $O(n\log(n)\log\log(n))$. More than 35 years later, Fürer (2007) developed an algorithm that ran in time $n\log(n)2^{\log^*(n)}$. In case you are wondering what that weird function $\log^*(n)$ (read "log star") is, it is the number of times you have to apply the logarithm function log() iteratively to n in order to get down to something ≤ 1. For all values of n less than the estimated number of atoms in the universe, the value of $\log^*(n)$ (with base 2) is less than 5. So $\log^*(n)$ is a really really *really* slowly growing function of n. Finally, Harvey and van der Hoeven (2019) gave an algorithm that runs in time $O(n\log n)$. This is conjectured to be optimal. It is quite amazing that the seemingly simple (and old) question of multiplying two numbers has proved to be so mysterious and has seen new research advances as recently as 2019. This is what makes the study of algorithms so exciting!

4 Etymology of the word "Algorithm"

As a round-about way of describing the etymology of the word "Algorithm", pause for a minute and consider how remarkable it is that 3rd graders can actually multiply large numbers. Its really amazing that anyone, let alone an 8-yr old, can multiply two 10-digit numbers. One reason multiplication is so easy for us is because we have a great data structure for numbers—we represent numbers using base-10 (Arabic) numerals, and this lends itself to easy arithmetic.

Why were romans so bad at multiplication? Well, imagine multiplying using roman numerals. What is LXXXIX times CM? The only way I can imagine computing this is to first translate the numbers into Arabic numerals $[LXXXIX = 50 + 10 + 10 + 10 + (-1) + 10 = 89$, and $CM = (-100) + 1000 = 900]$ then multiplying those the standard way. Roman numerals seem like a pretty lousy data structure if you want to do arithmetic.

The word "Algorithm" is a mangled transliteration of the name "al-Khwarizmi." Al-Khwarizmi was a 9th-century Persian polymath, born in present-day Uzbekistan, who studied and worked in Baghdad during the Abbassid Caliphate; around 820 AD he was appointed as the astronomer and head of the library of the House of Wisdom in Baghdad. He wrote several influential books, including one with the title [something like] "On the Calculation with Hindu Numerals", which described how to do arithmetic using Arabic numerals (aka Arabic-Hindu, or just Hindu numerals). The original manuscript was lost, though a Latin translation from the 1100's introduced this number system to Europe, and is responsible for why we use Arabic numerals today. (You can imagine how happy a 12th century tax collector would have been with this new ability to easily do arithmetic....) [The old French word *algorisme* meant "the Arabic numerals system", and only later did it come to mean a general recipe for solving computational problems.]

CS 161 (Stanford, Winter 2022) Lecture 2

Adapted From Virginia Williams' lecture notes. Additional credits: Michael Kim, Ofir Geri, Mary Wootters, Aviad Rubinstein.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

MergeSort, Recurrences 101, and Asymptotic Analysis

1 Introduction

In general, when analyzing an algorithm, we want to know two things.

1. Does it work?
2. Does it have good performance?

Today, we'll begin to see how we might formally answer these questions, through the lens of sorting. We'll start, as a warm-up, with InsertionSort, and then move on to the (more complicated) MergeSort. MergeSort will also allow us to continue our exploration of *Divide and Conquer.*

2 InsertionSort

In your pre-lecture exercise, you should have taken a look at a couple of ways of implementing InsertionSort. Here's one:

```
def InsertionSort(A):
   for i in range(1,len(A)):
      current = A[i]
      j = i-1
      while j >= 0 and A[j] > current:
          A[j+1] = A[j]
          j -= 1
      A[j+1] = current
```

Let us ask our two questions: does this algorithm work, and does it have good performance?

2.1 Correctness of InsertionSort

Once you figure out what InsertionSort is doing (see the slides for the intuition on this), you may think that it's “obviously" correct. However, if you didn't know what it was doing and just got the above code, maybe this wouldn't be so obvious. Additionally, for algorithms that we'll study in the future, it *won't* always be obvious that it works, and so we'll have to prove it. To warm us up for those proofs, let's carefully go through a proof of correctness of InsertionSort.

We'll do the proof by maintaining a *loop invariant,* in this case that after iteration i, then `A[:i+1]` is sorted. This is obviously true when $i = 0$ (because the one-element list $A[:1]$ is definitely sorted) and then we'll show that for any $i > 0$, if it's true for $i - 1$, then it's true for i. At the end of the day, we'll conclude that `A[:n]` (aka, the whole thing) is sorted and we'll be done.

Formally, we will proceed by induction.

- **Inductive hypothesis.** After iteration i of the outer loop, `A[:i+1]` is sorted.
- **Base case.** When $i = 0$, `A[:1]` contains only one element, and this is sorted.
- **Inductive step.** Suppose that the inductive hypothesis holds for $i - 1$, so `A[:i]` is sorted after the $i - 1$'st iteration. We want to show that `A[:i+1]` is sorted after the i'th iteration.

  Suppose that j^* is the largest integer in $\{0, \ldots, i - 1\}$ so that `A[j*] < A[i]`. Then the effect of the inner loop is to turn

  $$[A[0], A[1], \ldots, A[j^*], \ldots, A[i-1], A[i]]$$

  into

  $$[A[0], A[1], \ldots, A[j^*], A[i], A[j^*+1], \ldots, A[i-1]].$$

  We claim that this latter list is sorted. This is because $A[i] > A[j^*]$, and by the inductive hypothesis, we have $A[j^*] \geq A[j]$ for all $j \leq j^*$, and so $A[i]$ is larger than everything that is positioned before it. Similarly, by the choice of j^* we have $A[i] \leq A[j^*+1] \leq A[j]$ for all $j \geq j^* + 1$, so $A[i]$ is smaller than everything that comes after it. Thus, $A[i]$ is in the right place. All of the other elements were already in the right place, so this proves the claim.

  Thus, after the i'th iteration completes, `A[:i+1]` is sorted, and this establishes the inductive hypothesis for i.
- **Conclusion.** By induction, we conclude that the inductive hypothesis holds for all $i \leq n - 1$. In particular, this implies that after the end of the $n - 1$'st iteration (after the algorithm ends) `A[:n]` is sorted. Since `A[:n]` is the whole list, this means the whose list is sorted when the algorithm terminates, which is what we were trying to show.

The above proof was maybe a bit pedantic: we used a lot of words to prove something that may have been pretty obvious. However, it's important to understand the structure of this argument, because we'll use it a lot, sometimes for more complicated algorithms.

2.2 Running time of InsertionSort

The running time of InsertionSort is about n^2 operations. To be a bit more precise, at iteration i, the algorithm may have to look through and move i elements, so that's about $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ operations.

We're not going to stress the precise operation count, because we'll argue that the end of the lecture that we don't care too much about it. The main question that we have, is, can we do *asymptotically better* than n^2? That is, can we come up with an algorithm that sorts an arbitrary list of n integers in time that scales less than n^2? For example, like $n^{1.5}$, or $n \log(n)$, or even n? Next we'll see that MergeSort will scale like $n \log(n)$, which is much faster.

3 MergeSort

Recall the *Divide-and-conquer* paradigm from the first lecture. In this paradigm, we use the following strategy:

- Break the problem into sub-problems.
- Solve the sub-problems (often recursively)
- Combine the results of the sub-problems to solve the big problem.

At some point, the sub-problems become small enough that they are easy to solve, and then we can stop recursing.

With this approach in mind, MergeSort is a very natural algorithm to solve the sorting problem. The pseudocode is below:

```
MergeSort(A):
  n = len(A)
  if n <= 1:
     return A
  L = MergeSort( A[:n/2] )
  R = MergeSort( A[n/2:] )
  return Merge(L, R)
```

Above, we are using Python notation, so $A[: n/2] = [A[0], A[1], \ldots, A[n/2-1]]$ and $A[n/2 :] = [A[n/2], \ldots, A[n-1]]$. Additionally, we're using integer division, so `n/2` means $\lfloor n/2 \rfloor$.

How do we do the `Merge` procedure? We need to take two sorted arrays, L and R, and merge them into a sorted array that contains both of their elements. See the slides for a walkthrough of this procedure.

```
Merge(L, R):
   m = len(L) + len(R)
   S = [ ]
   for k in range(m):
      if L[i] < R[j]:
         S.append( L[i] )
         i += 1
      else:
         S.append( R[j] )
         j += 1
   return S
```

Note: This pseudocode is incomplete! What happens if we get to the end of L or R? Try to adapt the pseudocode above to fix this.

As before, we need to ask: Does it work? And does it have good performance?

3.1 Correctness of MergeSort

Let's focus on the first question first. As before, we'll proceed by induction. This time, we'll maintain a *recursion invariant* that any time MergeSort returns, it returns a sorted array.

- **Inductive Hypothesis.** Whenever MergeSort returns an array of size $\leq i$, that array is sorted.
- **Base case.** Suppose that $i = 1$. Then whenever MergeSort returns an array of length 0 or length 1, that array is sorted. (Since all array of length 0 and 1 are sorted). So the Inductive Hypothesis holds for $i = 1$.
- **Inductive step.** We need to show that if MergeSort always returns a sorted array on inputs of length $\leq i - 1$, then it always does for length $\leq i$. Suppose that MergeSort has an input of length i. Then L and R are both of length $\leq i - 1$, so by induction, L and R are both sorted. Thus, the inductive step boils down to the statement:

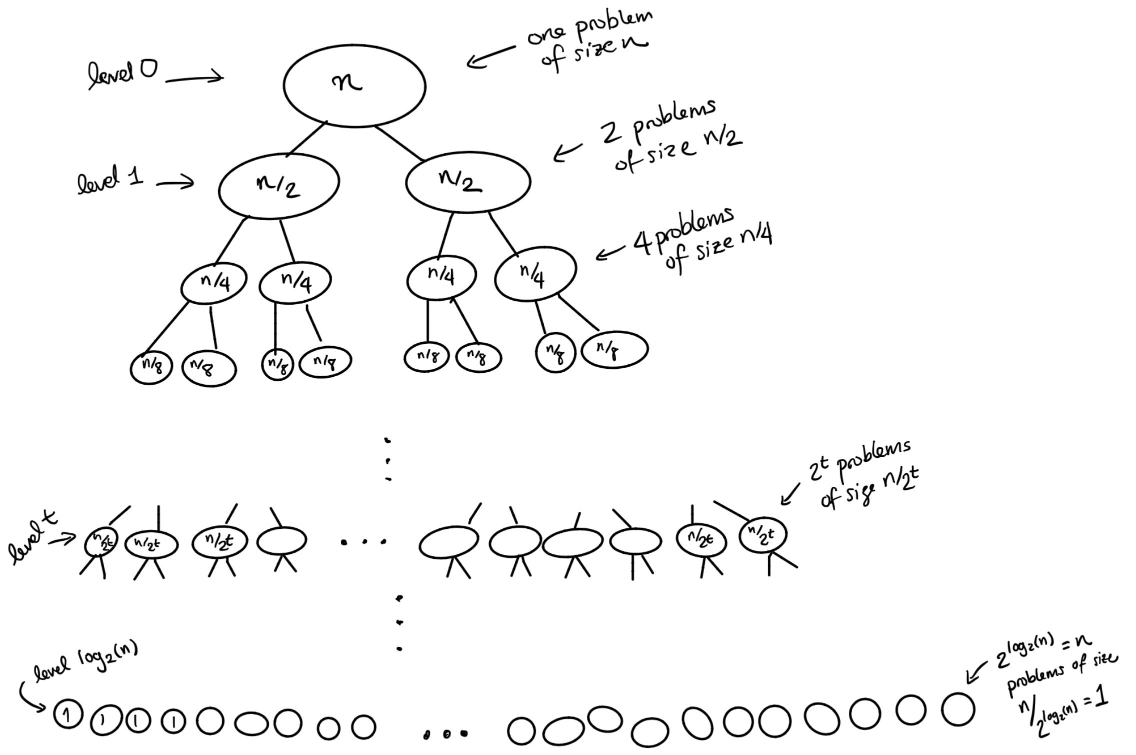
  "When Merge takes as inputs two sorted arrays L and R, then it returns a sorted array containing all of the elements of L, along with all of the elements of R."

  This statement is intuitively true, although proving it rigorously takes a bit of bookkeeping. In fact, it takes another proof by induction! Check out CLRS Section 2.3.1 for a rigorous treatment.
- **Conclusion.** By induction, the Inductive hypothesis holds for all i. In particular, given an array of any length n, MergeSort returns a sorted version of that array.

3.2 Running time of MergeSort

Finally, we get to our first question in this lecture where the answer may not be intuitively obvious. What is the running time of MergeSort? In the next few lectures, we'll see a few principled ways of analyzing the runtime of a recursive algorithm. Here, we'll just go through one of the ways, which is called the *recursion tree* method.

The idea is to draw a tree representing the computation (see the slides for the visuals). Each node in the tree represents a subproblem, and its children represent the subproblems we need to solve to solve the big sub-problem. The recursion tree for MergeSort looks something like this:



At the top (zeroth) level is the whole problem, which has size n. This gets broken into two sub-problems, each of size $n/2$, and so on. At the t'th level, there are 2^t problems, each of size $n/2^t$. This continues until we have n problems of size 1, which happens at the $\log(n)$'th level.

Some notes:

- In this class, logarithms will **always** be base 2, unless otherwise noted.
- We are being a bit sloppy in the picture above: what if n is not a power of 2? Then $n/2^j$ might not be an integer. In the pseudocode above, we actually break a problem of size n into problems of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Keeping track of this in our analysis will be messy, and it won't add much, so we will ignore it, and for now we will assume that n is a power of 2. [1]

[1] To formally justify the assumption that n is a power of 2, notice that we can always sort a *longer* list of length $n' = 2^{\lceil \log_2(n) \rceil}$. That is, we'll add extra entries, whose values are ∞, to the list. Then we sort the new list of length n', and return the first n values. Since $n' \leq 2n$ (why?) this won't affect the asymptotic running time. Also see CLRS Section 4.6.2 for a rigorous analysis of the original algorithm with floors and ceilings.

In order to figure out the total amount of work, we will figure out how much work is done at each node in the tree, and add it all up. To that end, we tally up the work that is done in a particular node in the tree—that is, in a particular call to MergeSort. There are three things:

1. Checking the base case
2. Making recursive calls (but we don't count the work actually done in those recursive calls; that will count in other nodes)
3. Running Merge.

Let's analyze each of these. Suppose that our input has size m (so that $m = n/2^j$ for some j).

1. Checking the base case doesn't take much time. For concreteness, let us say that it takes one operation to retrieve the length of A, and other operation to compare this length to 1, for a total of two operations.[2]

2. Making the recursive calls should also be fast. If we implemented the pseudocode well, it should also take a constant number of operations.

   > **Aside:** This is a good point to talk about how we interpret pseudocode in this class. Above, we've written `MergeSort(A[:n/2])` as an example of a recursive call. This makes it clear that we are supposed to recurse on the first half of the list, but it's not clear how we actually implement that. Our "pseudocode" above is in fact working Python code, and in Python, this implementation, while clear, is a bit inefficient. That is, written this way, Python will actually *copy* the first $n/2$ elements of the list before sending them to the recursive call. A much better way would be to instead just pass in pointers to the 0'th and $n/2-1$'st index in the list. This would result in a faster algorithm, but kludgier pseudocode. In this class, we generally will opt for cleaner pseudocode, as long as it does not hurt the *asymptotic* running time of the algorithm. In this case, our simple-but-slower pseudocode turns out not to affect the asymptotic running time, so we'll stick with this.

   In light of the above **Aside**, let's suppose that this step takes $m+2$ operations, $m/2$ to copy each half of the list over, and 2 operations to store the results. Of course, a better implementation of this step would only take a constant number (say, four) operations.

3. The third thing is the tricky part. We claim that the Merge step also takes about m operations.

   Consider a single call to Merge, where we'll assume the total size of A is m numbers. How long will it take for Merge to execute? To start, there are two initializations for i and j. Then, we enter a for loop which will execute m times. Each loop will require

[2] Of course, there's no reason that the "operation" of getting the length of A should take the same amount of time as the "operation" of comparing two integers. This disconnect is one of the reasons we'll introduce big-Oh notation at the end of this lecture.

one comparison, followed by an assignment to S and an increment of i or j. Finally, we'll need to increment the counter in the for loop k. If we assume that each operation costs us a certain amount of time, say $Cost_a$ for assignment, $Cost_c$ for comparison, $Cost_i$ for incrementing a counter, then we can express the total time of the Merge subroutine as follows:

$$2Cost_a + m(Cost_a + Cost_c + 2Cost_i)$$

This is a precise, but somewhat unruly expression for the running time. In particular, it seems difficult to keep track of lots of different constants, and it isn't clear which costs will be more or less expensive (especially if we switch programming languages or machine architectures). To simplify our analysis, we choose to assume that there is some global constant c_{op} which represents the cost of an operation. You may think of c_{op} as $\max\{Cost_a, Cost_c, Cost_i, \ldots\}$. We can then bound the amount of running time for Merge as

$$2c_{op} + 4c_{op}m = 2 + 4m \text{ operations}$$

Thus, the total number of operations is at *most*

$$2 + (m + 2) + 4m + 2 \leq 11m$$

using the assumption that $m \geq 1$. This is a very loose bound; for larger m this will be much closer to $5m$ than it is to $11m$. But, as we'll discuss more below, the difference between 5 and 11 won't matter too much to us, so much as the linear dependence on m.

Now that we understand how much work is going on in one call where the input has size m, let's add it all up to obtain a bound on the number of operations required for MergeSort. in a Merge of m numbers, we want to translate this into a bound on the number of operations required for MergeSort. At first glance, the pessimist in you may be concerned that at each level of recursive calls, we're spawning an exponentially increasing number of copies of MergeSort (because the number of calls at each depth doubles). Dual to this, the optimist in you will notice that at each level, the inputs to the problems are decreasing at an exponential rate (because the input size halves with each recursive call). Today, the optimists win out.

Claim 1. MergeSort *requires at most* $11n \log n + 11n$ *operations to sort* n *numbers.*

Before we go about proving this bound, let's first consider whether this running time bound is good. We mentioned earlier that more obvious methods of sorting, like InsertionSort, required roughly n^2 operations. How does $n^2 = n \cdot n$ compare to $n \cdot \log n$? An intuitive definition of $\log n$ is the following: "Enter n into your calculator. Divide by 2 until the total is ≤ 1. The number of times you divided is the logarithm of n." This number in general will be significantly smaller than n. In particular, if $n = 32$, then $\log n = 5$; if $n = 1024$, then $\log n = 10$. Already, to sort arrays of $\approx 10^3$ numbers, the savings of $n \log n$ as compared to n^2 will be orders of magnitude. At larger problem instances of 10^6, 10^9, etc. the difference will become even more pronounced! $n \log n$ is much closer to growing linearly (with n) than it is to growing quadratically (with n^2).

One way to argue about the running time of recursive algorithms is to use *recurrence relations*. A recurrence relation for a running time expresses the time it takes to solve an input of size n in terms of the time required to solve the recursive calls the algorithm makes. In particular, we can write the running time $T(n)$ for MergeSort on an array of n numbers as the following expression.

$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + T(\text{Merge}(n)) \\ &\leq 2 \cdot T(n/2) + 11n \end{aligned}$$

There are a number of sophisticated and powerful techniques for solving recurrences. We will cover many of these techniques in the coming lectures. Today, we can actually analyze the running time directly.

Proof of Claim 1. Consider the recursion tree of a call to MergeSort on an array of n numbers. Assume for simplicity that n is a power of 2. Let's refer to the initial call as Level 0, the proceeding recursive calls as Level 1, and so on, numbering the level of recursion by its depth in the tree. How deep is the tree? At each level, the size of the inputs is divided in half, and there are no recursive calls when the input size is ≤ 1 element. By our earlier "definition", this means the bottom level will be Level $\log n$. Thus, there will be a total of $\log n + 1$ levels.

We can now ask two questions: (1) How many subproblems are there at Level i? (2) How large are the individual subproblems at Level i? We can observe that at the ith level, there will be 2^i subproblems, each with inputs of size $n/2^i$.

We've already worked out that each sub-problem with input of size $n/2^i$ takes at most $11n/2^i$ operations. Now we can add this up:

$$\begin{aligned} \text{Work at Level } i &= (\text{number of subproblems}) \cdot (\text{work per subproblem}) \\ &\leq 2^i \cdot 11\left(\frac{n}{2^i}\right) \\ &= 11n \text{ operations.} \end{aligned}$$

Importantly, we can see that the work done at Level i is independent of i – it only depends on n and is the same for every level. This means we can bound the total running time as follows:

$$\begin{aligned} \text{Total number of operations} &= (\text{operations per level}) \cdot (\text{number of levels}) \\ &\leq (11n) \cdot (\log n + 1) \\ &= 11n \log n + 11n \end{aligned}$$

□

This proves the claim, and we're done!

4 Guiding Principles for Algorithm Design and Analysis

After going through the algorithm and analysis, it is natural to wonder if we've been too sloppy. In particular, note that the algorithm never "looks at" the input. For instance, what if we received the sequence of numbers $[1, 2, 3, 5, 4, 6, 7, 8]$? Clearly, there is a "sorting algorithm" for this sequence that only takes a few operations, but MergeSort runs through all $\log n + 1$ levels of recursion anyway. Would it be better to try to design our algorithms with this in mind? Additionally, in our analysis, we've given a very loose upper bound on the time required of Merge and dropped a number of constant factors and lower order terms. Is this a problem? In what follows, we'll argue that these are actually *features*, not bugs, in the design and analysis of the algorithm.

4.1 Worst-Case Analysis

One guiding principle we'll use throughout the class is that of *Worst-Case Analysis*. In particular, this means that we want any statement we make about our algorithms to hold for *every* possible input. Stated differently, we can think about playing a game against an adversary, who wants to maximize our running time (make it as bad as possible). We get to specify an algorithm and state a running time $T(n)$; the adversary then chooses an input. We win the game if even in the worst case, whatever input the adversary chooses (of size n), our algorithm runs in at most $T(n)$ time.

Note that because our algorithm made no assumptions about the input, then our running time bound will hold for every possible input. This is a very strong, robust guarantee. [3]

4.2 Asymptotic Analysis

Throughout our argument about MergeSort, we combined constants (think $Cost_a, Cost_i$, etc.) and gave very loose upper bounds (like being okay with a naive implementation of our pseudocode, or with this very wasteful upper bound $11m$ on the work done at a subproblem in MergeSort). Why did we choose to do this? First, it makes the math much easier. But does it come at the cost of getting the "right" answer? Would we get a more predictive result if we threw all these exact expressions back into the analysis? From the perspective of an algorithm designer, the answer is to both of these questions is a resounding "No". As an algorithm designer, we want to come up with results that are broadly applicable, whose truth does not depend on features of a specific programming language or machine architecture. The constants that we've dropped will depend greatly on the language and machine on which you're working. For the same reason we use pseudocode instead of writing our algorithms in Java, trying to quantify the exact running time of an algorithm would be inappropriately specific. This is not to say that constant factors never matter in applications (e.g. I would be rather upset if my web browser ran 7 times slower than it does now) but worrying about

[3] In the case where you have significant domain knowledge about which inputs are likely, you may choose to design an algorithm that works well in expectation on these inputs (this is frequently referred to as Average-Case Analysis). This type of analysis is often much more tricky, and requires strong assumptions on the input.

these factors is not the goal of this class. In this class, our goal will be to argue about which strategies for solving problems are wise and why.

In particular, we will focus on *Asymptotic Analysis*. This type of analysis focuses on the running time of your algorithm as your input size gets very large (i.e. $n \to +\infty$). This framework is motivated by the fact that if we need to solve a small problem, it doesn't cost that much to solve it by brute-force. If we want to solve a large problem, we may need to be much more creative in order for the problem to run efficiently. From this perspective, it should be very clear that $11n(\log n + 1)$ is much better than $n^2/2$. (If you are unconvinced, try plugging in some values for n.)

Intuitively, we'll say that an algorithm is "fast" when the running time grows "slowly" with the input size. In this class, we want to think of growing "slowly" as growing as close to linear as possible. Based on this this intuitive notion, we can come up with a formal system for analyzing how quickly the running time of an algorithm grows with its input size.

4.3 Asymptotic Notation

To talk about the running time of algorithms, we will use the following notation. $T(n)$ denotes the runtime of an algorithm on input of size n.

"Big-Oh" Notation:

Intuitively, Big-Oh notation gives an upper bound on a function. We say $T(n)$ is $O(f(n))$ when as n gets big, $f(n)$ grows at least as quickly as $T(n)$. Formally, we say

$$T(n) = O(f(n)) \iff \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,\ 0 \leq T(n) \leq c \cdot f(n)$$

"Big-Omega" Notation:

Intuitively, Big-Omega notation gives a lower bound on a function. We say $T(n)$ is $\Omega(f(n))$ when as n gets big, $f(n)$ grows at least as slowly as $T(n)$. Formally, we say

$$T(n) = \Omega(f(n)) \iff \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,\ 0 \leq c \cdot f(n) \leq T(n)$$

"Big-Theta" Notation:

$T(n)$ is $\Theta(f(n))$ if and only if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$. Equivalently, we can say that

$$T(n) = \Theta(f(n)) \iff \exists c_1, c_2, n_0 > 0 \text{ s.t. } \forall n \geq n_0,\ 0 \leq c_1 f(n) \leq T(n) \leq c_2 f(n)$$

We can see that these notations really do capture exactly the behavior that we want – namely, to focus on the rate of growth of a function as the inputs get large, ignoring constant factors and lower order terms. As a sanity check, consider the following example and non-example.

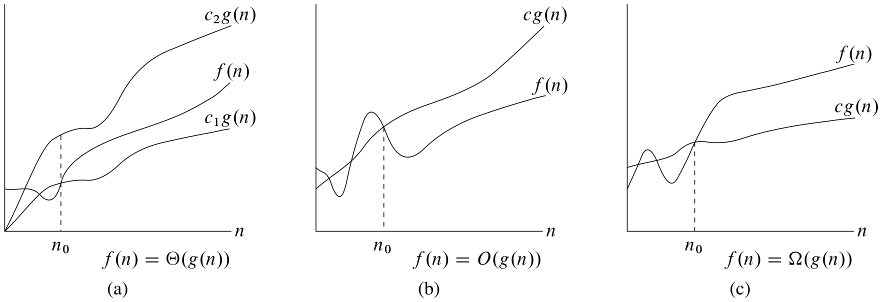


Figure 3.1 from CLRS – Examples of Asymptotic Bounds
(Note: In these examples $f(n)$ corresponds to our $T(n)$ and $g(n)$ corresponds to our $f(n)$.)

Claim 2. *All degree-k polynomials*[4] *are* $O(n^k)$.

Proof. Suppose $T(n)$ is a degree-k polynomial. That is, $T(n) = a_k n^k + \ldots + a_1 n + a_0$ for some choice of a_i's where $a_k \neq 0$. To show that $T(n)$ is $O(n^k)$ we must find a c and n_0 such that for all $n \geq n_0$ $T(n) \leq c \cdot n^k$. (Since $T(n)$ represents the running time of an algorithm, we assume it is positive.) Let $n_0 = 1$ and let $a^* = \max_i |a_i|$. We can bound $T(n)$ as follows:

$$\begin{aligned} T(n) &= a_k n^k + \ldots + a_1 n + a_0 \\ &\leq a^* n^k + \ldots + a^* n + a^* \\ &\leq a^* n^k + \ldots + a^* n^k + a^* n^k \\ &= (k+1) a^* \cdot n^k \end{aligned}$$

Let $c = (k+1)a^*$ which is a constant, independent of n. Thus, we've exhibited c, n_0 which satisfy the Big-Oh definition, so $T(n) = O(n^k)$. □

Claim 3. *For any* $k \geq 1$, n^k *is not* $O(n^{k-1})$.

Proof. By contradiction. Assume $n^k = O(n^{k-1})$. Then there is some choice of c and n_0 such that $n^k \leq c \cdot n^{k-1}$ for all $n \geq n_0$. But this in turn means that $n \leq c$ for all $n \geq n_0$, which contradicts the fact that c is a constant, independent of n. Thus, our original assumption was false and n^k is not $O(n^{k-1})$. □

[4] To be more precise, all degree-k polynomials T so that $T(n) \geq 0$ for all $n \geq 1$. How would you adapt this proof to be true for all degree-k polynomials T with a positive leading coefficient?

CS 161 (Stanford, Winter 2022) Lecture 3

Adapted From Virginia Williams' lecture notes. Additional credits: Albert Chen, Juliana Cook, Ofir Geri, Sam Kim, Gregory Valiant, Aviad Rubinstein.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

Solving Recurrences and the Selection Problem

1 Introduction

Today we will continue to talk about divide and conquer, and go into detail on how to solve recurrences.

Recall that divide and conquer algorithms divide up a problem into a number of subproblems that are the smaller instances of the same problem, solve those problems recursively, and combine the solutions to the subproblems into a solution for the original problem. When a subproblem size is small enough, the subproblem is solved in a straightforward manner. In the past lectures we have seen two examples of divide and conquer algorithms: MergeSort and Karatsuba's algorithm for integer multiplication.

The running time of divide and conquer algorithms can be naturally expressed in terms of the running time of smaller inputs. Today we will show two techniques for solving these recurrences. The first is called the *master method* to solve these recurrences. This method can only be used when the size of all the subproblems is the same (as was the case in the examples). We will also see a surprising algorithm that does not fall into this category, and how to analyze its running time using another method, the *substitution method*.

2 Recurrences

Stated more technically, a divide and conquer algorithm takes an input of size n and does some operations all running in $O(f(n))$ time for some f and runs itself recursively on $k \geq 1$ instances of size $n_1, n_2, ..., n_k$, where $n_i < n$ for all i. To talk about what the runtime of such an algorithm is, we can write a runtime **recurrence**. Recurrences are functions defined in terms of themselves with smaller arguments, as well as one or more base cases. We can define a recurrence more formally as follows:

Let $T(n)$ be the worst-case runtime on instances of size n. If we have k recursive calls on a given step (of sizes n_i) and each step takes time $O(f(n))$, then we can write the runtime as $T(n) \leq c \cdot f(n) + \sum_{i=1}^{k} T(n_i)$ for some constant c, where our base case is $T(c') \leq O(1)$.

Now let's try finding recurrences for some of the divide and conquer algorithms we have seen.

2.1 Integer Multiplication

Recall the integer multiplication problem, where we are given two n-digit integers x and y and output the product of the two numbers. The long multiplication/grade school algorithm runs in $O(n^2)$ time. In lecture 1 we saw two divide and conquer algorithms for solving this problem. In both of them, we divided each of x and y into two $(n/2)$-digit numbers in the following way: $x = 10^{\frac{n}{2}}a + b$ and $y = 10^{\frac{n}{2}}c + d$. Then we compute $xy = ac \cdot 10^n + 10^{\frac{n}{2}}(ad + bc) + bd$.

In the first algorithm, which we call Mult1, we simply computed the four products ac, ad, bc, bd. Karatsuba found that since we only need the sum of ad and bc, we can save one multiplication operation by noting that $ad + bc = (a + b)(c + d) - ac - bd$.

Algorithm 1: Mult1(x, y)

```
Split $x$ and $y$ into $x = 10^{\frac{n}{2}}a + b$ and $y = 10^{\frac{n}{2}}c + d$
$z_1 = \text{Mult1}(a,\ c)$
$z_2 = \text{Mult1}(a,\ d)$
$z_3 = \text{Mult1}(b,\ c)$
$z_4 = \text{Mult1}(b,\ d)$
return $z_1 \cdot 10^n + 10^{\frac{n}{2}}(z_2 + z_3) + z_4$
```

Algorithm 2: Karatsuba(x, y)

```
Split $x = 10^{\frac{n}{2}}a + b$ and $y = 10^{\frac{n}{2}}c + d$
$z_1 = \text{Karatsuba}(a,\ c)$
$z_2 = \text{Karatsuba}(b,\ d)$
$z_3 = \text{Karatsuba}(a + b,\ c + d)$
$z_4 = z_3 - z_1 - z_2$
return $z_1 \cdot 10^n + z_4 \cdot 10^{\frac{n}{2}} + z_2$
```

We now express the running time of these two algorithms using recurrences. Adding two n digit integers is an $O(n)$ operation, since for each position we add at most three digits: the ith digit from each number and possibly a carry from the additions due to the $(i-1)$th digits.

Let $T_1(n)$ and $T_2(n)$ denote the worst-case runtime of Mult1 and Karatsuba, respectively, on inputs of size n. Then, the runtime of Mult1 can be written as the recurrence

$$T_1(n) = 4T_1\left(\frac{n}{2}\right) + O(n),$$

and Karatsuba's runtime can be written as the recurrence

$$T_2(n) = 3T_2\left(\frac{n}{2}\right) + O(n).$$

Note that the constant "hidden" in the $O(n)$ term in T_2 may be greater than in T_1, but for asymptotic analysis of the running time, these constants are not important.

2.2 MergeSort

Consider the basic steps for algorithm MergeSort(A), where $|A| = n$.

1. If $|A| = 1$, return A.
2. Split A into A_1, A_2 of size $\frac{n}{2}$.
3. Run MergeSort(A_1) and MergeSort(A_2).
4. Merge(A_1, A_2)

Steps 2 and 4 each take time $O(n)$. In step 3, we are splitting the work up into two subproblems of size $\frac{n}{2}$. Therefore, we get the following recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n).$$

In the previous lecture, we saw that the running time of MergeSort is $O(n \log n)$. In this lecture we will show how to derive this using the master method.

3 The Master Method

We now introduce a general method, called the *master method*, for solving recurrences where all the subproblems are of the same size. We assume that the input to the master method is a recurrence of the form

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d).$$

In this recurrence, there are three constants:

- a is the number of subproblems that we create from one problem, and must be an integer greater than or equal to 1.
- b is the factor by which the input size shrinks (it must hold that $b > 1$).
- d is the exponent of n in the time it takes to generate the subproblems and combine their solutions.

There is another constant “hidden" in the big-O notation. We will introduce it in the proof and see that it does not affect the result.

In addition, we need to specify the “base case" of the recurrence, that is, the runtime when the input gets small enough. For a sufficiently small n (say, when $n = 1$), the worst-case runtime of the algorithm is constant, namely, $T(n) = O(1)$.

We now state the *master theorem*, which is used to solve the recurrences.

Theorem 1 (Master Theorem)**.** *Let $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$ be a recurrence where $a \geq 1, b > 1$. Then,*

$$T(n) = \begin{cases} O(n^d \log n) & \textit{if } a = b^d \\ O(n^d) & \textit{if } a < b^d \\ O(n^{\log_b a}) & \textit{if } a > b^d \end{cases}$$

Remark 2. In some cases, the recurrence may involve subproblems of size $\lceil \frac{n}{b} \rceil$, $\lfloor \frac{n}{b} \rfloor$, or $\frac{n}{b} + 1$. The master theorem holds for these cases as well. However, we do not prove that here.

Before we turn to the proof of the master theorem, we show how it can be used to solve the recurrences we saw earlier.

- Mult1: $T(n) = 4T\left(\frac{n}{2}\right) + O(n)$.

  The parameters are $a = 4, b = 2, d = 1$, so $a > b^d$, hence $T(n) = O(n^{\log_2 4}) = O(n^2)$.

- Karatsuba: $T(n) = 3T\left(\frac{n}{2}\right) + O(n)$.

  The parameters are $a = 3, b = 2, d = 1$, so $a > b^d$, hence $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$.

- MergeSort: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$.

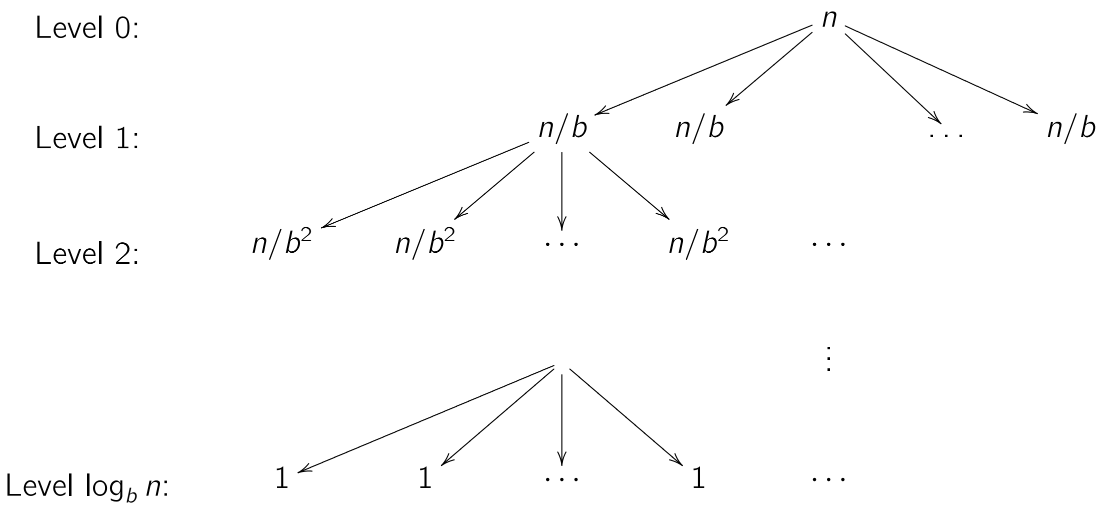
  The parameters are $a = 2, b = 2, d = 1$, so $a = b^d$, hence $T(n) = O(n \log n)$.

- Another example: $T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$.

  The parameters are $a = 2, b = 2, d = 2$, so $a < b^d$, hence $T(n) = O(n^2)$.

We see that for integer multiplication, Karatsuba is the clear winner!

Proof of the Master Theorem. Let $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$ be the recurrence we solve using the master theorem. For simplicity, we assume that $T(1) = 1$ and that n is a power of b. From the definition of big-O, we know that there is a constant $c > 0$ such that for sufficiently large n, $T(n) \leq a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$. The proof of the master theorem will use the recursion tree in a similar way to our analysis of the running time of MergeSort.



The recursion tree drawn above has $\log_b n + 1$ level. We analyze the amount of work done at each level, and then sum over all levels in order to get the total running time. Consider level j. At level j, there are a^j subproblems. Each of these subproblems is of size $\frac{n}{b^j}$, and will take time at most $c\left(\frac{n}{b^j}\right)^d$ to solve (this only considers the work done at level j and does not include the time it takes to solve the subsubproblems). We conclude that the total work done at level j is at most $a^j \cdot c\left(\frac{n}{b^j}\right)^d = cn^d\left(\frac{a}{b^d}\right)^j$.

Writing the running time this way shows us where the terms a and b^d come from: a is the branching factor and measures how the number of subproblems grows at each level, and b^d is the shrinkage in the work needed (per subproblem).

Summing over all levels, we get that the total running time is at most $cn^d \sum_{j=0}^{log_b n}\left(\frac{a}{b^d}\right)^j$. We now consider each of the three cases.

1. $a = b^d$. In this case, the amount of work done at each level is the same: cn^d. Since there are $\log_b n+1$ levels, the total running time is at most $(\log_b n+1)cn^d = O(n^d \log n)$.
2. $a < b^d$. In this case, $\frac{a}{b^d} < 1$, hence, $\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j \leq \sum_{j=0}^{\infty}\left(\frac{a}{b^d}\right)^j = \frac{1}{1-\frac{a}{b^d}} = \frac{b^d}{b^d-a}$. Hence, the total running time is $cn^d \cdot \frac{b^d}{b^d-a} = O(n^d)$.

   Intuitively, in this case the shrinkage in the work needed per subproblem is more significant, so the work done in the highest level “dominates" the other factors in the running time.
3. $a > b^d$. In this case, $\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j = \frac{\left(\frac{a}{b^d}\right)^{\log_b n+1}-1}{\frac{a}{b^d}-1}$. Since a, b, c, d are constants, we get that the total work done is $O\left(n^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n}\right) = O\left(n^d \cdot \frac{a^{\log_b n}}{b^{d \log_b n}}\right) = O\left(n^d \cdot \frac{n^{\log_b a}}{n^d}\right) = O(n^{\log_b a})$.

   Intuitively, here the branching factor is more significant, so the total work done at each level increases, and the leaves of the tree “dominate".

□

We conclude with a more general version of the master theorem.

Theorem 3 (Master Theorem - more general version)**.** *Let $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ be a recurrence where $a \geq 1$, $b > 1$. Then,*

- *If $f(n) = O\left(n^{\log_b(a)-\epsilon}\right)$ for some constant $\epsilon > 0$, $T(n) = \Theta\left(n^{\log_b(a)}\right)$.*
- *If $f(n) = \Theta\left(n^{\log_b(a)}\right)$, $T(n) = \Theta\left(n^{\log_b a} \log n\right)$.*
- *If $f(n) = \Omega\left(n^{\log_b(a)+\epsilon}\right)$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

4 The Substitution Method

Recurrence trees can get quite messy when attempting to solve complex recurrences. With the substitution method, we can guess what the runtime is, plug it in to the recurrence and see if it works out.

Given a recurrence $T(n) \leq f(n) + \sum_{i=1}^{k} T(n_i)$, we can guess that the solution to the recurrence is

$$T(n) \leq \begin{cases} d \cdot g(n_0) \text{ if } n = n_0 \\ d \cdot g(n) \text{ if } n > n_0 \end{cases}$$

for some constants $d > 0$ and $n_0 \geq 1$ and a function $g(n)$. We are essentially guessing that $T(n) \leq O(g(n))$.

For our base case we must show that you can pick some d such that $T(n_0) \leq d \cdot g(n_0)$. For example, this can follow from our standard assumption that $T(1) = 1$.

Next we assume that our guess is correct for everything smaller than n, meaning $T(n') \leq d \cdot g(n')$ for all $n' < n$. Using the inductive hypothesis, we prove the guess for n. We must pick some d such that

$$f(n) + \sum_{i=1}^{k} d \cdot g(n_i) \leq d \cdot g(n), \text{whenever } n \geq n_0.$$

Typically the way this works is that you first try to prove the inductive step starting from the inductive hypothesis, and then from this obtain a condition that d needs to obey. Using this condition you try to figure out the base case, i.e., what n_0 should be.

5 Selection

The selection problem is to find the kth smallest number in an array A.

Input: array A of n numbers, and an integer $k \in \{1, \cdots, n\}$.

Output: the k-th smallest number in A.

One approach is to sort the numbers in ascending order, and then return the kth number in the sorted list. This takes $O(n \log n)$ time, since it takes $O(n \log n)$ time for the sort (e.g. by MergeSort) and $O(1)$ time to return kth number.

5.1 Minimum Element

As always, we ask if we can do better (i.e. faster in big-O terms). In the special case where $k = 1$, selection is the problem of finding the minimum element. We can do this in $O(n)$ time by scanning through the array and keeping track of the minimum element so far. If the current element is smaller than the minimum so far, we update the minimum.

Algorithm 3: SelectMin(A)

```
$m \leftarrow \infty$
$n \leftarrow \text{length}(A)$
for $i = 1$ to $n$ do
    if $A(i) < m$ then
        $m \leftarrow A(i)$
return $m$
```

In fact, this is the best running time we could hope for.

Definition 4. A deterministic algorithm is one which, given a fixed input, always performs the same operations (as opposed to an algorithm which uses randomness).

Claim 5. *Any deterministic algorithm for finding the minimum has runtime $\Omega(n)$.*

Proof of Claim 5. Intuitively, the claim holds because any algorithm for the minimum must look at all the elements, each of which could be the minimum. Suppose a correct deterministic algorithm does not look at $A(i)$ for some i. Then the output cannot depend on $A(i)$, so the algorithm returns the same value whether $A(i)$ is the minimum element or the maximum element. Therefore the algorithm is not always correct, which is a contradiction. So there is no sublinear deterministic algorithm for finding the minimum. □

So for $k = 1$, we have an algorithm which achieves the best running time possible. By similar reasoning, this lower bound of $\Omega(n)$ applies to the general selection problem. So ideally we would like to have a linear-time selection algorithm in the general case.

6 Linear-Time Selection

In fact, a linear-time selection algorithm does exist. Before showing the linear time selection algorithm, it's helpful to build some intuition on how to approach the problem. The high-level idea will be to try to do a Binary Search over an unsorted input. At each step, we hope to divide the input into two parts, the subset of smaller elements of A, and the subset of larger elements of A. We will then determine whether the kth smallest element lies in the first part (with the "smaller" elements) or the part with larger elements, and recurse on exactly one of those two parts.

How do we decide how to partition the array into these two pieces? Suppose we have a black-box algorithm ChoosePivot that chooses some element in the array A, and we use this pivot to define the two sets–any $A[i]$ less than the pivot is in the set of "smaller" values, and any $A[i]$ greater than the pivot is in the other part. We will figure out precisely how to specify this subroutine ChoosePivot a bit later, after specifying the high-level algorithm structure. For clarity we'll assume all elements are distinct from now on, but the idea generalizes easily. Let n be the size of the array and assume we are trying to find the k^{th} element.

Algorithm 4: Select(A, n, k)

```
if $n == 1$ then
    return $A[1]$
$p \leftarrow$ ChoosePivot($A, n$)
$A_< \leftarrow \{A(i) \mid A(i) < p\}$
$A_> \leftarrow \{A(i) \mid A(i) > p\}$
if $|A_<| = k - 1$ then
    return $p$
else if $|A_<| > k - 1$ then
    return Select($A_<, |A_<|, k$)
else if $|A_<| < k - 1$ then
    return Select($A_>, |A_>|, k - |A_<| - 1$)
```

At each iteration, we use the element p to partition the array into two parts: all elements smaller than the pivot and all elements larger than the pivot, which we denote $A_<$ and $A_>$, respectively.

Depending on what the size of the resulting sub-arrays are, the runtime can be different. For example, if one of these sub-arrays is of size $n - 1$, at each iteration, we only decreased the size of the problem by 1, resulting in total running time $O(n^2)$. If the array is split into two equal parts, then the size of the problem at iteration reduces by half, resulting in a linear time solution. (We assume ChoosePivot runs in $O(n)$.)

Claim 6. *If the pivot p is chosen to be the minimum or maximum element, then* Select *runs in* $\Theta(n^2)$ *time.*

Proof. At each iteration, the number of elements decreases by 1. Since running ChoosePivot and creating $A_<$ and $A_>$ takes linear time, the recurrence for the runtime is $T(n) = T(n-1) + \Theta(n)$. Expanding this,

$$T(n) \leq c_1 n + c_1(n-1) + c_1(n-2) + ... + c_1 = c_1 n(n+1)/2$$

and

$$T(n) \geq c_2 n + c_2(n-1) + c_2(n-2) + ... + c_2 = c_2 n(n+1)/2.$$

We conclude that $T(n) = \Theta(n^2)$. □

Claim 7. *If the pivot p is chosen to be the median element, then* Select *runs in $O(n)$ time.*

Proof. Intuitively, the running time is linear since we remove half of the elements from consideration each iteration. Formally, each recursive call is made on inputs of half the size, namely, $T(n) \leq T(n/2) + cn$. Expanding this, the runtime is $T(n) \leq cn + cn/2 + cn/4 + ... + c \leq 2cn$, which is $O(n)$. □

So how do we design ChoosePivot that chooses a pivot in linear time? In the following, we describe three ideas.

6.1 Idea #1: Choose a random pivot

As we saw earlier, depending on the pivot chosen, the worst-case runtime can be $O(n^2)$ if we are unlucky in the choice of the pivot at every iteration. As you might expect, it is extremely unlikely to be this unlucky, and one can prove that the *expected* runtime is $O(n)$ provided the pivot is chosen uniformly at random from the set of elements of A. In practice, this randomized algorithm is what is implemented, and the hidden constant in the $O(n)$ runtime is very small.

6.2 Idea #2: Choose a pivot that creates the most “balanced" split

Consider ChoosePivot that returns the pivot that creates the most “balanced" split, which would be the median of the array. However, this is exactly selection problem we are trying to solve, with $k = n/2$! As long as we do not know how to find the median in linear time, we cannot use this procedure as ChoosePivot.

6.3 Idea #3: Find a pivot "close enough" to the median

Given a linear-time median algorithm, we can solve the selection problem in linear time (and vice versa). Although ideally we would want to find the median, notice that as far as correctness goes, there was nothing special about partitioning around the median. We could use this same idea of partitioning and recursing on a smaller problem even if we partition around an arbitrary element. To get a good runtime, however, we need to guarantee that the subproblems get smaller quickly. In 1973, Blum, Floyd, Pratt, Rivest, and Tarjan came up with the Median of Medians algorithm. It is similar to the previous algorithm, but rather than partitioning around the exact median, uses a surrogate “median of medians". We update ChoosePivot accordingly.

Algorithm 5: ChoosePivot(A, n)

```
Split $A$ into $g = \lceil n/5 \rceil$ groups $p_1, \ldots, p_g$
for $i = 1$ to $g$ do
    $p_i \leftarrow$ MergeSort($p_i$)
$C \leftarrow \{$median of $p_i \mid i = 1, \ldots, g\}$
$p \leftarrow$ Select($C, g, g/2$)
return $p$
```

What is this algorithm doing? First it divides A into segments of size 5. Within each group, it finds the median by first sorting the elements with MergeSort. Recall that MergeSort sorts in $O(n \log n)$ time. However, since each group has a constant number of elements, it takes constant time to sort. Then it makes a recursive call to Select to find the median of C, the median of medians. Intuitively, by partitioning around this value, we are able to find something that is close to the true median for partitioning, yet is ‘easier’ to compute, because it is the median of $g = \lceil n/5 \rceil$ elements rather than n. The last part is as before: once we have our pivot element p, we split the array and recurse on the proper subproblem, or halt if we found our answer.

We have devised a slightly complicated method to determine which element to partition around, but the algorithm remains correct for the same reasons as before. So what is its running time? As before, we're going to show this by examining the size of the recursive subproblems. As it turns out, by taking the median of medians approach, we have a guarantee on how much smaller the problem gets each iteration. The guarantee is good enough to achieve $O(n)$ runtime.

6.3.1 Running Time

Lemma 8. $|A_<| \leq 7n/10 + 5$ *and* $|A_>| \leq 7n/10 + 5$.

Proof of Lemma 8. p is the median of $p_1, \cdots, p_g$. Because p is the median of $g = \lceil n/5 \rceil$ elements, the medians of $\lceil g/2 \rceil - 1$ groups p_i are smaller than p. If p is larger than a group median, it is larger than at least three elements in that group (the median and the smaller two numbers). This applies to all groups except the remainder group, which might have fewer than 5 elements. Accounting for the remainder group, p is greater than at least $3 \cdot (\lceil g/2 \rceil - 2)$ elements of A. By symmetry, p is less than at least the same number of elements.

Now,

$$\begin{aligned} |A_>| &= \# \text{ of elements greater than } p \\ &\leq (n-1) - 3\cdot(\lceil g/2\rceil - 2) \\ &= n + 5 - 3\cdot\lceil g/2\rceil \\ &\leq n - 3n/10 + 5 \\ &= 7n/10 + 5. \end{aligned} \tag{1}$$

By symmetry, $|A_<| \leq 7n/10 + 5$ as well.

Intuitively, we know that 60% of half of the groups are less than the pivot, which is 30% of the total number of elements, n. Therefore, at most 70% of the elements are greater than the pivot. Hence, $|A_>| \approx 7n/10$. We can make the same argument for $|A_<|$. □

The recursive call used to find the median of medians has input of size $\lceil n/5\rceil \leq n/5 + 1$. The other work in the algorithm takes linear time: constant time on each of $\lceil n/5\rceil$ groups for MergeSort (linear time total for that part), $O(n)$ time scanning A to make $A_<$ and $A_>$.

Thus, we can write the full recurrence for the runtime,

$$T(n) \leq \begin{cases} c_1 n + T(n/5+1) + T(7n/10+5) & \text{if } n > 5 \\ c_2 & \text{if } n \leq 5. \end{cases}$$

How do we prove that $T(n) = O(n)$? The master theorem does not apply here. Instead, we will prove this using the substitution method.

6.4 Solving the Recurrence of Select using the Substitution Method

For simplicity, we consider the recurrence $T(n) \leq T(n/5) + T(7n/10) + cn$ instead of the exact recurrence of Select.

To prove that $T(n) = O(n)$, we guess:

$$T(n) \leq \begin{cases} dn_0 \text{ if } n = n_0 \\ d\cdot n \text{ if } n > n_0 \end{cases}$$

For the base case, we pick $n_0 = 1$ and use the standard assumption that $T(1) = 1 \leq d$. For the inductive hypothesis, we assume that our guess is correct for any $n < k$, and we prove our guess for k. That is, consider d such that for all $n_0 \leq n < k$, $T(n) \leq dn$.

To prove for $n = k$, we solve the following equation:

$$T(k) \leq T(k/5) + T(7k/10) + ck \leq dk/5 + 7dk/10 + ck \leq dk$$

$$9/10d + c \leq d$$

$$c \leq d/10$$

$$d \geq 10c$$

Therefore, we can choose $d = max(1, 10c)$, which is a constant factor. The induction is completed. By the definition of big-O, the recurrence runs in $O(n)$ time.

6.5 Issues when using the Substitution Method

Now we will try out an example where our guess is incorrect. Consider the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + n$ (similar to MergeSort). We will guess that the algorithm is linear.

$$T(n) \leq \begin{cases} dn_0 \text{ if } n = n_0 \\ d \cdot n \text{ if } n > n_0 \end{cases}$$

We try the inductive step. We try to pick some d such that for all $n \geq n_0$,

$$n + \sum_{i=1}^{k} dg(n_i) \leq d \cdot g(n)$$

$$n + 2 \cdot d \cdot \frac{n}{2} \leq dn$$

$$n(1 + d) \leq dn$$

$$n + dn \leq dn$$

$$n < 0,$$

However, the above can never be true, and there is no choice of d that works! Thus our guess was incorrect.

This time the guess was incorrect since MergeSort takes superlinear time. Sometimes, however, the guess can be asymptotically correct but the induction might not work out. Consider for instance $T(n) \leq 2T(n/2) + 1$.

We know that the runtime is $O(n)$ so let's try to prove it with the substitution method. Let's guess that $T(n) \leq cn$ for all $n \geq n_0$.

First we do the induction step: We assume that $T(n/2) \leq cn/2$ and consider $T(n)$. We want that $2 \cdot cn/2 + 1 \leq cn$, that is, $cn + 1 \leq cn$. However, this is impossible.

This doesn't mean that $T(n)$ is not $O(n)$, but in this case we chose the wrong linear function. We could guess instead that $T(n) \leq cn - 1$. Now for the induction we get $2 \cdot (cn/2 - 1) + 1 = cn - 1$ which is true for all c. We can then choose the base case $T(1) = 1$.

CS 161 (Stanford, Winter 2022) Lecture 4

Adapted from Virginia Williams' lecture notes and Mary Wootters' handouts. Additional credits: Albert Chen, Juliana Cook, Ofir Geri, Sam Kim, Gregory Valiant, Aviad Rubinstein. Please direct all typos and mistakes to Moses Charikar and Nima Anari.

Median and Selection

1 Introduction

So far we have covered the master theorem, which can be used for recurrences of a certain form. Recall that if we have a recurrence $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$ where $a \geq 1, b > 1$, then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Many algorithms that result from the divide-and-conquer paradigm yield recurrence relations for their runtimes that have the above form – namely algorithms that divide the problem into equal-sized sub-pieces at each recursion.

Today, we will introduce a problem where the master theorem cannot be applied: the problem of finding the k-th smallest element in an unsorted array. First, we show it can be done in $O(n \log n)$ time via sorting and that any correct algorithm must run in $\Omega(n)$ time. However, it is not obvious that a linear-time selection algorithm exists. We present a linear-time selection algorithm, with an intuition for why it has the desired properties to achieve $O(n)$ running time. The two high-level goals of this lecture are 1) to cover a really cool and surprising algorithm, and 2) illustrate that some divide-and-conquer algorithms yield recurrence relations that cannot be analyzed via the "Master Method/Theorem", yet one can (often) still successfully analyze them.

2 Selection

The selection problem is to find the k-th smallest number in an array A.

Input: array A of n numbers, and an integer $k \in \{1, \ldots, n\}$.

Output: the k-th smallest number in A.

One approach is to sort the numbers in ascending order, and then return the k-th number in the sorted list. This takes $O(n \log n)$ time, since it takes $O(n \log n)$ time for the sort (e.g. by MergeSort) and $O(1)$ time to return k-th number.

2.1 Minimum Element

As always, we ask if we can do better (i.e., faster in big-O terms). In the special case where $k = 1$, selection is the problem of finding the minimum element. We can do this in $O(n)$ time by scanning through the array and keeping track of the minimum element so far. If the current element is smaller than the minimum so far, we update the minimum.

Algorithm 1: SelectMin(A)

```
$m \leftarrow \infty$
$n \leftarrow \text{length}(A)$
for $i = 1$ to $n$ do
    if $A[i] < m$ then
        $m \leftarrow A[i]$
return $m$
```

In fact, this is the best running time we could hope for.

Definition 1. A deterministic algorithm is one which, given a fixed input, always performs the same operations (as opposed to an algorithm which uses randomness).

Proposition 2. *Any deterministic algorithm for finding the minimum has runtime $\Omega(n)$.*

Proof. Intuitively, the claim holds because any algorithm for the minimum must look at all the elements, each of which could be the minimum. Suppose a correct deterministic algorithm does not look at $A[i]$ for some i. Then the output cannot depend on $A[i]$, so the algorithm returns the same value whether $A[i]$ is the minimum element or the maximum element. Therefore the algorithm is not always correct, which is a contradiction. So there is no sublinear deterministic algorithm for finding the minimum. □

So for $k = 1$, we have an algorithm which achieves the best running time possible. By similar reasoning, this lower bound of $\Omega(n)$ applies to the general selection problem. So ideally we would like to have a linear-time selection algorithm in the general case.

3 Linear-Time Selection

In fact, a linear-time selection algorithm does exist. Before showing the linear time selection algorithm, it's helpful to build some intuition on how to approach the problem. The high-level idea will be to try to do a Binary Search over an unsorted input. At each step, we hope to divide the input into two parts, the subset of smaller elements of A, and the subset of larger elements of A. We will then determine whether the k-th smallest element lies in the first part (with the "smaller" elements) or the part with larger elements, and recurse on exactly one of those two parts.

How do we decide how to partition the array into these two pieces? Suppose we have a black-box algorithm ChoosePivot that chooses some element in the array A, and we use this pivot to define the two sets–any $A[i]$ less than the pivot is in the set of “smaller” values, and any $A[i]$ greater than the pivot is in the other part. We will figure out precisely how to specify this subroutine ChoosePivot a bit later, after specifying the high-level algorithm structure. The algorithm ChoosePivot does not affect the *correctness* of the algorithm as we will see in Section 3.6. Rather, it only affects the runtime.

For clarity we'll assume all elements are distinct from now on, but the idea generalizes easily. Let n be the size of the array and assume we are trying to find the k-th element.

Algorithm 2: Select(A, n, k)

```
if $n = 1$ then
    return $A[1]$
$p \leftarrow$ ChoosePivot($A, n$)
$A_< \leftarrow \{A(i) \mid A(i) < p\}$
$A_> \leftarrow \{A(i) \mid A(i) > p\}$
if $|A_<| = k - 1$ then
    return $p$
else if $|A_<| > k - 1$ then
    return Select($A_<, |A_<|, k$)
else if $|A_<| < k - 1$ then
    return Select($A_>, |A_>|, k - |A_<| - 1$)
```

At each iteration, we use the element p to partition the array into two parts: all elements smaller than the pivot and all elements larger than the pivot, which we denote $A_<$ and $A_>$, respectively.

Depending on what the size of the resulting sub-arrays are, the runtime can be different. For example, if one of these sub-arrays is of size $n-1$, at each iteration, we only decreased the size of the problem by 1, resulting in total running time $O(n^2)$. If the array is split into two equal parts, then the size of the problem at iteration reduces by half, resulting in a linear time solution. (We assume ChoosePivot runs in $O(n)$.)

Proposition 3. *If the pivot p is chosen to be the minimum or maximum element, then* Select *runs in $\Theta(n^2)$ time.*

Proof. At each iteration, the number of elements decreases by 1. Since running ChoosePivot and creating $A_<$ and $A_>$ takes linear time, the recurrence for the runtime is $T(n) = T(n-1) + \Theta(n)$. Expanding this,

$$T(n) \leq c_1 n + c_1(n-1) + c_1(n-2) + \ldots + c_1 = c_1 n(n+1)/2$$

and

$$T(n) \geq c_2 n + c_2(n-1) + c_2(n-2) + ... + c_2 = c_2 n(n+1)/2.$$

We conclude that $T(n) = \Theta(n^2)$. □

Proposition 4. *If the pivot p is chosen to be the median element, then* Select *runs in $O(n)$ time.*

Proof. Intuitively, the running time is linear since we remove half of the elements from consideration each iteration. Formally, each recursive call is made on inputs of half the size, namely, $T(n) \leq T(n/2) + cn$. Expanding this, the runtime is $T(n) \leq cn + cn/2 + cn/4 + ... + c \leq 2cn$, which is $O(n)$. □

So how do we design ChoosePivot that chooses a pivot in linear time? In the following, we describe three ideas.

3.1 Idea #1: Choose a random pivot

As we saw earlier, depending on the pivot chosen, the worst-case runtime can be $O(n^2)$ if we are unlucky in the choice of the pivot at every iteration. As you might expect, it is extremely unlikely to be this unlucky, and one can prove that the *expected* runtime is $O(n)$ provided the pivot is chosen uniformly at random from the set of elements of A. In practice, this randomized algorithm is what is implemented, and the hidden constant in the $O(n)$ runtime is very small.

3.2 Idea #2: Choose a pivot that creates the most "balanced" split

Consider ChoosePivot that returns the pivot that creates the most "balanced" split, which would be the median of the array. However, this is exactly selection problem we are trying to solve, with $k = n/2$! As long as we do not know how to find the median in linear time, we cannot use this procedure as ChoosePivot.

3.3 Idea #3: Find a pivot "close enough" to the median

Given a linear-time median algorithm, we can solve the selection problem in linear time (and vice versa). Although ideally we would want to find the median, notice that as far as correctness goes, there was nothing special about partitioning around the median. We could use this same idea of partitioning and recursing on a smaller problem even if we partition around an arbitrary element. To get a good runtime, however, we need to guarantee that the subproblems get smaller quickly. In 1973, Blum, Floyd, Pratt, Rivest, and Tarjan came up with the Median of Medians algorithm. It is similar to the previous algorithm, but rather than partitioning around the exact median, uses a surrogate "median of medians". We update ChoosePivot accordingly.

Algorithm 3: ChoosePivot(A, n)

```
Split $A$ into $g = \lceil n/5 \rceil$ groups $p_1, \dots, p_g$
for $i = 1$ to $g$ do
    $p_i \leftarrow$ MergeSort($p_i$)
$C \leftarrow \{$median of $p_i \mid i = 1, \dots, g\}$
$p \leftarrow$ Select($C, g, g/2$)
return $p$
```

What is this algorithm doing? First it divides A into segments of size 5. Within each group, it finds the median by first sorting the elements with MergeSort. Recall that MergeSort sorts in $O(n \log n)$ time. However, since each group has a constant number of elements, it takes constant time to sort. Then it makes a recursive call to Select to find the median of C, the median of medians. Intuitively, by partitioning around this value, we are able to find something that is close to the true median for partitioning, yet is 'easier' to compute, because it is the median of $g = \lceil n/5 \rceil$ elements rather than n. The last part is as before: once we have our pivot element p, we split the array and recurse on the proper subproblem, or halt if we found our answer.

We have devised a slightly complicated method to determine which element to partition around, but the algorithm remains correct for the same reasons as before. So what is its running time? As before, we're going to show this by examining the size of the recursive subproblems. As it turns out, by taking the median of medians approach, we have a guarantee on how much smaller the problem gets each iteration. The guarantee is good enough to achieve $O(n)$ runtime.

3.3.1 Running Time

Lemma 5. *$|A_<| \leq 7n/10 + 5$ and $|A_>| \leq 7n/10 + 5$.*

Proof. p is the median of $p_1, \cdots, p_g$. Because p is the median of $g = \lceil n/5 \rceil$ elements, the medians of $\lceil g/2 \rceil - 1$ groups p_i are smaller than p. If p is larger than a group median, it is larger than at least three elements in that group (the median and the smaller two numbers). This applies to all groups except the remainder group, which might have fewer than 5 elements. Accounting for the remainder group, p is greater than at least $3 \cdot (\lceil g/2 \rceil - 2)$ elements of A. By symmetry, p is less than at least the same number of elements.

Now,

$$\begin{aligned} |A_>| &= \# \text{ of elements greater than } p \\ &\leq (n-1) - 3 \cdot (\lceil g/2 \rceil - 2) \\ &= n + 5 - 3 \cdot \lceil g/2 \rceil \\ &\leq n - 3n/10 + 5 \\ &= 7n/10 + 5. \end{aligned} \tag{1}$$

By symmetry, $|A_<| \leq 7n/10 + 5$ as well.

Intuitively, we know that 60% of half of the groups are less than the pivot, which is 30% of the total number of elements, n. Therefore, at most 70% of the elements are greater than the pivot. Hence, $|A_>| \approx 7n/10$. We can make the same argument for $|A_<|$. □

The recursive call used to find the median of medians has input of size $\lceil n/5 \rceil \leq n/5 + 1$. The other work in the algorithm takes linear time: constant time on each of $\lceil n/5 \rceil$ groups for MergeSort (linear time total for that part), $O(n)$ time scanning A to make $A_<$ and $A_>$.

Thus, we can write the full recurrence for the runtime,

$$T(n) \leq \begin{cases} c_1 n + T(n/5 + 1) + T(7n/10 + 5) & \text{if } n > 5 \\ c_2 & \text{if } n \leq 5. \end{cases}$$

How do we prove that $T(n) = O(n)$? The master theorem does not apply here. Instead, we will prove this using the substitution method.

3.4 Solving the Recurrence of Select Using the Substitution Method

For simplicity, we consider the recurrence $T(n) \leq T(n/5) + T(7n/10) + cn$ instead of the exact recurrence of Select.

To prove that $T(n) = O(n)$, we guess:

$$T(n) \leq \begin{cases} d \cdot n_0 & \text{if } n = n_0 \\ d \cdot n & \text{if } n > n_0 \end{cases}$$

For the base case, we pick $n_0 = 1$ and use the standard assumption that $T(1) = 1 \leq d$. For the inductive hypothesis, we assume that our guess is correct for any $n < k$, and we prove our guess for k. That is, consider d such that for all $n_0 \leq n < k$, $T(n) \leq dn$.

To prove for $n = k$, we solve the following equation:

$$T(k) \leq T(k/5) + T(7k/10) + ck \leq dk/5 + 7dk/10 + ck \leq dk$$

$$9/10d + c \leq d$$

$$c \leq d/10$$

$$d \geq 10c$$

Therefore, we can choose $d = max(1, 10c)$, which is a constant factor. The induction is completed. By the definition of big-O, the recurrence runs in $O(n)$ time.

3.5 Issues When Using the Substitution Method

Now we will try out an example where our guess is incorrect. Consider the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + n$ (similar to MergeSort). We will guess that the algorithm is linear.

$$T(n) \leq \begin{cases} d \cdot n_0 & \text{if } n = n_0 \\ d \cdot n & \text{if } n > n_0 \end{cases}$$

We try the inductive step. We try to pick some d such that for all $n \geq n_0$,

$$n + \sum_{i=1}^{k} dg(n_i) \leq d \cdot g(n)$$

$$\begin{gathered} n + 2 \cdot d \cdot \frac{n}{2} \leq dn \\ n(1+d) \leq dn \\ n + dn \leq dn \\ n < 0, \end{gathered}$$

However, the above can never be true, and there is no choice of d that works! Thus our guess was incorrect.

This time the guess was incorrect since MergeSort takes superlinear time. Sometimes, however, the guess can be asymptotically correct but the induction might not work out. Consider for instance $T(n) \leq 2T(n/2) + 1$.

We know that the runtime is $O(n)$ so let's try to prove it with the substitution method. Let's guess that $T(n) \leq cn$ for all $n \geq n_0$.

First we do the induction step: We assume that $T(n/2) \leq cn/2$ and consider $T(n)$. We want that $2 \cdot cn/2 + 1 \leq cn$, that is, $cn + 1 \leq cn$. However, this is impossible.

This doesn't mean that $T(n)$ is not $O(n)$, but in this case we chose the wrong linear function. We could guess instead that $T(n) \leq cn - 1$. Now for the induction we get $2 \cdot (cn/2 - 1) + 1 = cn - 1$ which is true for all c. We can then choose the base case $T(1) = 1$.

3.6 Correctness of the Algorithm

Recall that the choice of pivot only affects the runtime, and not the correctness of the algorithm. Here, we prove formally, by induction, that Select is correct. We will use *strong induction*. That is, our inductive step will assume that the inductive hypothesis holds for *all* n between 1 and $i - 1$, and then we'll show that it holds for $n = i$.

Remark 6. You can also do this using regular induction with a slightly more complicated inductive hypothesis; either way is fine.

Inductive Hypothesis (for n). When run on an array A of size n and an integer $k \in \{1, \ldots, n\}$, Select returns the k-th smallest element of A.

Base Case ($n = 1$). When $n = 1$, the requirement $k \in \{1, \ldots, n\}$ means that $k = 1$; that is, Select(A, k) is supposed to return the smallest element of A. This is precisely what the pseudocode above does when $|A| = 1$, so this establishes the Inductive Hypothesis for $n = 1$.

Inductive Step. Let $i \geq 2$, and suppose that the inductive hypothesis holds for all n with $1 \leq n < i$. Our goal is to show that it holds for $n = i$. That is, we would like to show that

> When run on an array A of size i and an integer $k \in \{1, \ldots, i\}$, Select(A, k) returns the k-th smallest element of A.

Informally, we want to show that assuming that Select "works" on smaller arrays, then it "works" on an array of length n.

We do this below:

Suppose that $1 \leq k \leq i$, and that A is an array of length i. There are three cases to consider, depending on $p =$ ChoosePivot(A, i). Notice that in the pseudocode above, p is a value from A, not an index. Let $A_<, A_>, p$ be as in the pseudocode above.

- **Case 1.** Suppose that $|A_<| = k - 1$. Then by the definition of $A_<$, there are $k - 1$ elements of A that are smaller than p, so p must be the k-th smallest. In this case, we return p, which is indeed the k-th smallest.
- **Case 2.** Suppose that $|A_<| > k - 1$. Then there are more than $k - 1$ elements of A that are smaller than p, and so in particular the k-th smallest element of A is the same as the k-th smallest element of L. Next we will use the inductive hypothesis for $n = |A_<|$, which holds since $|A_<| < i$. Since $1 \leq k \leq |A_<|$, the inductive hypothesis implies that Select$(A_<, k)$ returns the k-th smallest element of $A_<$. Thus, by returning this we are also returning the k-th smallest element of A, as desired.
- **Case 3.** Suppose that $|A_<| < k - 1$. Then there are fewer than $k - 1$ elements that are less than p, which means that the k-th smallest element of A must be greater than p; that is, it shows up in $A_>$. Now, the k-th smallest element in A is the same as the $(k - |A_<| - 1)$-st element in $A_>$. To see this, notice that there are $|A_<| + 1$ elements smaller than the k-th that do *not* show up in $A_>$. Thus there are $k - (|A_<| + 1) = k - |A_<| - 1$ elements in $A_>$ that are smaller than or equal to the k-th element. Now we want to apply the inductive hypothesis for $n = |A_>|$, which we can do since $|A_>| < i$. Notice that we have $1 \leq k - |A_<| - 1 \leq |A_>|$; the first inequality holds because $k > |A_<| + 1$ by the definition of Case 3, and the second inequality holds because it is the same as $k \leq |A_<| + |A_>| + 1 = n$, which is true by assumption. Thus, the inductive hypothesis implies that Select$(A_>, k - |A_<| - 1)$ returns the $(k - |A_<| - 1)$-st element of $A_>$. Thus, by returning this we are also returning the k-th smallest element of A, as desired.

Thus, in each of the three cases, Select(A, k) returns the k-th smallest element of A. This establishes the inductive hypothesis for $n = i$.

Conclusion. By induction, the inductive hypothesis holds for *all* $n \geq 1$. Thus, we conclude that Select(A, k) returns the k-th smallest element of A on any array A, provided that $k \in \{1, \ldots, |A|\}$. That is, Select is correct, which is what we wanted to show.

CS 161 (Stanford, Winter 2022) Lecture 5

Adapted from Virginia Williams' lecture notes. Additional credits go to Sam Keller, Seth Hildick-Smith, Gregory Valiant.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

Randomized Algorithms and Quicksort

Today we will study another sorting algorithm: Quicksort, which was invented in 1959 by Tony Hoare. You may wonder why we want to study a new sorting algorithm. We have already studied MergeSort, which we showed to perform significantly better than the trivial $O(n^2)$ algorithm. While MergeSort achieves an $O(n \log n)$ worst-case asymptotic bound, in practice, there are a number of implementation details about MergeSort that make it tricky to achieve high performance. Quicksort is an alternative algorithm, which is simpler to implement in practice. Quicksort will also use a divide and conquer strategy but will use randomization to improve the performance of the algortithm in expectation. Java, Unix, and C stdlib all have implementations of Quicksort as one of their built-in sorting routines.

1 Quicksort Overview

As in all sorting algorithms, we start with an array A of n numbers; again we assume without loss of generality that the numbers are distinct[1]. Quicksort is very similar to the Select algorithm we studied last lecture. The description of Quicksort is the following:

```
if $length(A) \leq 1$ then
    return $A$ // It is trivially sorted
else
    Pick some element $x \leftarrow A[i]$. // We call $x$ the pivot.
    Split $A$ into $A_< = \{A[i] \mid A[i] < x\}$ and $A_> = \{A[i] \mid A[i] > x\}$.
    Rearrange $A$ into $[A_<, x, A_>]$.
    Recurse on $A_<$ and $A_>$.
```

The above steps define a "partition" function on A. The partition function of Quicksort can vary depending on how the pivot is chosen and also on the implementation. Quicksort is often used in practice because we can implement this step in linear time, and with very small constant factors. In addition, the rearrangement (Step 3) can be done in-place, rather than making several copies of the array (as is done in MergeSort). In these notes we will not describe the details of an in-place implementation, but the pseudocode can be found in CLRS.

[1] What does it mean to say that we can assume something without loss of generality? Why can we make this assumption without loss of generality in this proof?

2 Speculation on the Runtime

The performance of Quicksort depends on which element is chosen as the pivot. Assume that we choose the k^{th} smallest element; then $|A_<| = k-1, |A_>| = n-k$.

This allows us to write a recurrence; let $T(n)$ be the runtime of Quicksort on an n-element array. We know the partition step takes $O(n)$ time; therefore the recurrence is

$$T(n) \leq cn + T(k-1) + T(n-k).$$

For the worst pivot choice (the maximum or minimum element in the array[2]), the runtime satisfies the recurrence $T(n) = T(n-1) + O(n)$; hence $T(n) = O(n^2)$.

One way that seems optimal to define the partition function is to pick the *median* as the pivot. In the above recurrence this would mean that $k = \lceil \frac{n}{2} \rceil$. We showed in the previous lecture that the algorithm Select can find the median element in linear time. Therefore the recurrence becomes $T(n) \leq cn + 2T(\frac{n}{2})$. This is exactly the same recurrence as MergeSort, which means that this algorithm is guaranteed to run in $O(n \log n)$ time.

Unfortunately, the median selection algorithm is not practical; while it runs in linear time, it has much larger constant factors than we would like. To improve it, we will explore some alternative methods for choosing a pivot.

We leave the proof of correctness of Quicksort as an exercise to the reader (hint: use induction).

2.1 Random Pivot Selection

Our discussion so far suggests two approaches to pivot selection: (1) We could pick an arbitrary element as a pivot. This is super simple and takes $O(1)$ time (good), but the worst case running time of the resulting algorithm would be $O(n^2)$ (bad). (2) We could pick the median as the pivot. This takes $O(n)$ time and is somewhat complicated (bad), but gives a worst case running time of $O(n \log n)$ (good). Can we get the best of both worlds?

One method of "defending" against making a bad pivot choice is to choose a random element as the pivot. We observe that it is unlikely that the random element will be either the median (best-case) or the maximum or minimum (worst-case). Note that we have a uniform distribution over the n order statistics of the array (this is a fancy way of saying: for every $1 \leq i \leq n$ we pick the i-th highest element with the same probability $\frac{1}{n}$). Let's pause for a moment. This is the first time in this course that we have encountered an algorithm that uses randomness in its execution. Such an algorithm (that "flips coins" and takes actions based on the outcome of these coin flips) is called a **randomized algorithm**. How do we analyze a randomized algorithm?

[2] Why are these the worst pivot choices?

3 Worst-Case Analysis

In this section we will derive a bound on the worst-case running time of Quicksort. If we consider the worst random choice of pivot at each step, the running time will be $\Theta(n^2)$. This flavor of worst-case analysis (which gets an upper bound on the running time over all possible possible choices of pivots) is no different from the worst case analysis of the algorithm which picks an arbitrary pivot at every step. We are thus interested in what the running time of Quicksort is **on average over all possible choices of the pivots**. We should emphasize an important point: We still consider the running time for a **worst-case input**, and **average** only **over the random choices of the algorithm** (which is different from averaging over all possible inputs). Put differently, our analysis will guarantee that **for any input, the expected running time will be small**. We formalize the idea of averaging over the random choices of the algorithm by considering the running time of the algorithm on an input I as a random variable and bounding the expectation of that random variable.

Proposition 1. *For every input array of size n, the expected running time of Quicksort is $O(n \log n)$.*

Recall that a random variable is a function that maps every element in the sample space to a real number. In the case of rolling a die, the real number (or value of the point in the sample space) would be the number on the top of the die. Here the sample space is the set of all possible choices of pivots, and an example for a random variable can be the running time of Quicksort on a specific input I.

Denote by z_i the i-th element in the sorted array. For each i, j, we define a random variable $X_{i,j}(\sigma)$ to be the number of times z_i and z_j are compared for a given series of pivot choices σ. What are the possible values for $X_{i,j}(\sigma)$? It can be 0 if z_i and z_j are not compared. Note that all comparisons are with the pivot, and that the pivot is not included in the elements of the arrays in the recursive calls. If z_i and z_j are compared, consider the first time that this happens: one of them must be the pivot at this stage and this pivot is excluded from the subarrays that recursive calls operate on. Thus, no two elements are compared twice. Therefore, $X_{i,j}(\sigma) \in \{0, 1\}$.

Our goal is to compute the expected number of comparisons that Quicksort makes. Recall the definition of expectation:

$$\mathbb{E}[X] = \sum_{\sigma} \mathbb{P}[\sigma] X(\sigma) = \sum_{k} k \, \mathbb{P}[X = k].$$

Unfortunately for us, this definition does not really give us any clue about how we can actually go about computing the expectation of the complicated random variable that we have at hand, i.e. the number of comparisons made by Quicksort. Can you imagine trying to figure out the probability that Quicksort actually performs exactly k comparisons? We don't have the foggiest idea how we might do that. If you do, please let us know!

Luckily for us, there is way around this. An important property of expectation is **linearity of**

expectation. For any random variables $X_1, \ldots, X_n$:

$$\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i].$$

This is a really simple to state, but amazingly useful property that makes the computation of expectations of seemingly complicated random variables much *much* easier. It is worth internalizing this technique that we are about to apply for analyzing the expected number of comparisons that Quicksort performs.

We start with computing the expected value of $X_{i,j}$. These variables are **indicator random variables**, which take the value 1 if some event happens, and 0 otherwise. The expected value is

$$\begin{aligned}\mathbb{E}[X_{i,j}] &= \mathbb{P}[X_{i,j} = 1] \cdot 1 + \mathbb{P}[X_{i,j} = 0] \cdot 0 \\ &= \mathbb{P}[X_{i,j} = 1]\end{aligned}$$

Let $C(\sigma)$ be the total number of comparisons made by Quicksort for a given set of pivot choices σ:

$$C(\sigma) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{i,j}(\sigma).$$

We wish to compute $\mathbb{E}[C]$ to get the expected number of comparisons made by Quicksort for an input array of size n.

$$\begin{aligned}\mathbb{E}[C] &= \mathbb{E}\left[\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{i,j}(\sigma)\right] \\ &= \sum_{i=1}^{n} \mathbb{E}\left[\sum_{j=i+1}^{n} X_{i,j}(\sigma)\right] \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{P}[z_i, z_j \text{ are compared}]\end{aligned}$$

Now we find $\mathbb{P}[z_i, z_j \text{ are compared}]$. Apriori, this seems difficult to do, but it turns out that there is a really elegant way to analyze this. Note that each element in the array (except the pivot) is compared to the pivot at each level of the recurrence. To analyze $\mathbb{P}[z_i, z_j \text{ are compared}]$, examine the portion of the array $[z_i \ldots., z_j]$. After the array is split using a pivot from $[z_i \ldots., z_j]$, z_i and z_j can no longer be compared. Hence, z_i and z_j are compared only when from the portion of the array $[z_i \ldots., z_j]$, either z_i or z_j is the first one picked as the pivot. So,

$$\begin{aligned}\mathbb{P}[z_i, z_j \text{ compared}] &= \mathbb{P}[z_i \text{ or } z_j \text{ is the first pivot picked from } [z_i \ldots., z_j]] \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\ &= \frac{2}{j-i+1}\end{aligned}$$

Make sure you are able to explain the second line in the calculation above.

We return to the expected value of C:

$$\begin{aligned}\mathbb{E}[C] &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} \mathbb{P}[z_i, z_j \text{ are compared}] \\ &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{2}{j-i+1}\end{aligned}$$

Note that for a fixed i,

$$\begin{aligned}\sum_{j=i+1}^{n} \frac{1}{j-i+1} &= \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-i+1} \\ &\leq \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}\end{aligned}$$

And using $\sum_{k=2}^{n} \frac{1}{k} \leq \ln n$, we get that

$$\begin{aligned}\mathbb{E}[C] &= \mathbb{E}[\sum_{i=1}^{n}\sum_{j=i+1}^{n} X_{i,j}(\sigma)] \\ &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &\leq 2n \ln n\end{aligned}$$

Thus, the expected number of comparisons made by Quicksort is no greater than $2n \ln n = O(n \log n)$. To complete the proof, we have to show that the running time is dominated by the number of comparisons. Note that in each recursive call to Quicksort on an array of size k, the algorithm performs $k-1$ comparisons in order to split the array, and the amount of work done is $O(k)$. In addition, Quicksort will be called on single-element arrays at most once for each element in the original array, so the total running time of Quicksort is $O(C+n)$. In conclusion, the expected running time of Quicksort on worst-case input is $O(n \log n)$.

3.1 Alternative Proof

Here we provide an alternative method for bounding the expected number of comparisons. Let $T(n)$ be the *expected* number of comparisons performed by Quicksort on an input of size n. In general, if the pivot is chosen to be the i-th order statistic of the input array (i.e. the ith largest element),

$$T(n) = n - 1 + T(i-1) + T(n-i).$$

where we define $T(0) = 0$. Each of the n possible choices of i are equally likely. Thus, the expected number of comparisons is:

$$\begin{aligned} T(n) &= n - 1 + \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i)) \\ &= n - 1 + \frac{2}{n}\sum_{i=1}^{n-1}(T(i)) \end{aligned}$$

Did you catch this minor detail: why does the second summation go from 1 through $n-1$, while on the previous line, the summation goes from 1 through n?

Continuing the calculation, we will use two facts:

1. $\sum_{i=1}^{n-1} f(i) \leq \int_1^n f(x)dx$ for an increasing function f.
   (How would you prove this? This gives us a way to upper-bound discrete sums by integrals of continuous functions – a nice trick to have in your toolkit that we will use below. If you understand this, what is the corresponding statement to lower-bound a discrete sum by an integral? What if f was a decreasing function of n? It is good to get into the habit of asking yourself such questions to test your understanding.)
2. $\int 2x \ln x dx = x^2 \ln x - \frac{x^2}{2} + C$.

Now we show that $T(n) \leq 2n \ln n$ by (strong) induction.

Inductive Hypothesis. $T(i) \leq 2i \ln i$.

Base case ($i = 1$) An array of size 1 requires no comparisons. Thus, $T(1) = 0$ and the inductive hypothesis is true for $i = 1$.

Inductive step. We will show that if the inductive hypothesis is true for all $i \leq k - 1$ then the inductive hypothesis is also true for $i = k$.

Let's bound $T(k)$:

$$\begin{aligned}
T(k) &= k - 1 + \frac{2}{k}\sum_{i=1}^{k-1} T(i) \\
&\le k - 1 + \frac{2}{k}\sum_{i=1}^{k-1} 2i\ln i \\
&\le k - 1 + \frac{2}{k}\int_1^k (2x\ln x)dx \\
&= k - 1 + \frac{2}{k}\left[k^2\ln k - \frac{k^2}{2} + \frac{1}{2}\right] \\
&= 2k\ln k + k - 1 - k + \frac{1}{k} \\
&= 2k\ln k - 1 + \frac{1}{k} \\
&\le 2k\ln k.
\end{aligned}$$

Thus the inductive hypothesis is true for $i = k$. This establishes the inductive step.

Inductive step. By induction, we have proved that $T(i) \le 2i\ln i$ for all i. Hence $T(n) \le 2n\ln n$. This concludes the proof.

CS 161 (Stanford, Winter 2022) Lecture 6

Adapted from Virginia Williams' lecture notes. Additional credits go to William Chen and Gregory Valiant.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

Sorting Lower Bounds,
Counting Sort, and Radix Sort

1 Lower Bounds for Comparison-Based Sorting Algorithms

See link on the course website.

2 Counting Sort

Recall that our sorting lower bounds applied to the class of algorithms that can only evaluate the values being sorted via *comparison queries*, namely via asking whether a given element is greater than, less than, or equal to some other element. For such algorithms, we proved that any correct algorithm (even a randomized one!) will require $\Omega(n \log n)$ such queries on some input. As with any lower-bound that we prove in a restricted model, it is fruitful to ask "is it possible to have an algorithm that does not fall in this class, and hence is not subject to the lowerbound?" For sorting, the answer is "yes".

We start by looking at a very simple non-comparison-based sorting algorithm called Counting Sort. Since it is not comparison-based, it is not restricted by the $\Omega(n \log n)$ lower bound for sorting. For a given input of n objects, each with a corresponding key (or value) in the range $\{0, 1, ..., r-1\}$, Counting Sort will sort the objects by their keys:

1. Create an array A of r buckets where each bucket contains a linked list.
2. For each element in the input array with key k, concatenate the element to the end of the linked list $A[k]$.
3. Concatenate all the linked lists: $A[0], \ldots, A[r-1]$.

The algorithm correctly sorts the n elements by their keys because the elements are placed into buckets by key where bucket i (containing elements with key = i) will come before bucket j (containing elements with key = j) in A for $i < j$. Therefore, when the algorithm concatenates the buckets, all elements with key = i will come before elements with key = j.

The worst case run time of Counting sort is O($n+r$) since it performs $O(1)$ passes over the n input elements and $O(1)$ passes over the r buckets of A. This is great if the size of the range r is small, but we pay dearly in the running time (and space) if r is large. The algorithm we discuss in the next section builds on Counting Sort and fixes this issue (that Counting Sort behaves poorly if the range of values r is very large).

An important property (which we will use in the next section) is that the algorithm described above is *stable*: If two input elements x, y have the same key, and x appears before y in the input array, x will appear before y in the output.

3 Radix Sort

Radix Sort is another non-comparison-based sorting algorithm that will use Counting Sort as a subroutine. Assuming that the input array contains d-digit numbers where each digit ranges from 0 to $r-1$, Radix Sort sorts the array digit-by-digit (or field-by-field for non-numerical inputs). The algorithm works on input array A as follows:

```
1. For $j = 1, \ldots d$:
2.       Apply Counting Sort to A using the $j^{\text{th}}$ digit as the key.
```

Note that we refer to the least significant digit as the first digit. Hence, the algorithm calls Counting Sort first using the least significant digit as the key, then again using the second least significant digit, until the most significant digit.

We will show that Radix Sort correctly sorts an input list of n numbers via induction on the iterations of the loop.

Inductive Hypothesis: At the end of the j^{th} iteration, the elements in A are sorted when considering only the j least significant digits of each element.

Base case: ($j = 1$) Radix Sort correctly sorts the numbers by the first digit as it uses Counting Sort to sort the numbers using the least significant digit as a key. (Note: we could have used $j = 0$ as the base case as well).

Inductive case: We will prove that, if the inductive hypothesis for $j = k - 1$ is true, then the inductive hypothesis for $j = k$ is true as well.

By the inductive hypothesis for $j = k - 1$, by the end of iteration $k - 1$, the input numbers have been sorted by the $k - 1$ least significant digits. After we run Counting Sort on the elements using digit k as the key, the numbers are sorted by their j^{th} digit. Since Counting Sort is stable, the elements in each bucket keep their original order, and by the induction hypothesis, they are ordered by their $k - 1$ least significant digits. Since **the elements are ordered first by their k^{th} digit, and then by their $k-1$ least significant digits**, we conclude that they are ordered by their k least significant digits.

To provide more detail for the bolded text in that last sentence, we could consider any two numbers x and y in the input and separately consider two cases as we did in lecture: case (i) where x and y have different kth least significant digits, and case (ii) where x and y have identical kth least significant digits. In both cases, we could establish that, in the output,

x and y are ordered by their least k significant digits. Since this is true for every pair of numbers x and y in the input, this means that the output of this iteration is correctly sorted by the k least significant digits.

By induction, the inductive hypothesis is true for all $j \in \{1, \ldots, d\}$. Applying the inductive hypothesis for $j = d$, we conclude that at the end of iteration d, the numbers are in sorted order (since the input consists of d-digit numbers). In other words, Radix Sort correctly sorts the input.

The worst case running time of Radix Sort is O($d(n + r)$) since we are calling Counting Sort on n elements with r possible keys once for each digit in the input numbers. If r = O(n) and d = O(1), then this takes O(n) time.

As we did in lecture, we could consider varying the base r in which we write our numbers and running radix sort (with the Counting Sort in the inner loop using r buckets). Now the maximum number of digits is a function of the maximum size of the numbers in the input and our choice of base r. If we have an input consisting of n numbers of value at most M, then the number of digits $d = \lfloor \log_r(M) \rfloor + 1$. (verify this yourself!) Thus the running time for Radix Sort with base r is $O(d(n + r))$ which is $O((\lfloor \log_r(M) \rfloor + 1) \cdot (n + r))$.

How should we choose r? One reasonable choice is $r = n$ to balance the two values n and r in the term $(n + r)$ in the expression for the running time. For this choice of r, the running time for Radix sort is $O(n \cdot (\lfloor \log_r(M) \rfloor + 1))$. (Why didn't we drop the $+1$ term in the expression for the asymptotic running time?) Note that we might make a different choice of r if it was also important to optimize the space required to run Radix Sort.

So the running time of Radix Sort (in terms of n) depends on how large the numbers in the input are as a function of n. If the upper bound $M \leq n^c$ for some constant c, then the running time is $O(n)$. In this regime, Radix Sort beats MergeSort and Quicksort. On the other hand, if $M = 2^n$, then the running time is $O(n^2 / \log n)$. In this regime, we would not use Radix Sort.

CS 161 (Stanford, Winter 2022) Lecture 7

Adapted from Virginia Williams' lecture notes. Additional credits go to Ilan Goodman, Vishnu Sundaresan, Wilbur Yang and Gregory Valiant.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

Heaps and Binary Search Trees

1 Data Structures

Thus far in this course we have mainly discussed algorithm design, and have specified algorithms at a relatively high level. For example, in describing sorting algorithms, we often assumed that we could insert numbers into a list in constant time; we didn't worry too much about the actual implementation of this, and took it as an assumption that this could actually be implemented (e.g. via a linked-list).

In this lecture, we will talk about the actual design and implementation of some useful data-structures. The purpose of this lecture is mainly to give you a taste of data-structure design. We won't discuss this area too much more in the course, and will switch back to discussing higher-level algorithms. If you like data-structures, CS166 is an entire course about them!

To motivate the data structures that we will discuss in this lecture, consider the following table that lists a bunch of basic operations that we would like to perform on a set/list of numbers, together with the worst-case runtime of performing those operation for two of the data structures that you are (hopefully) familiar with: linked lists of n (unsorted) numbers, and an array of n sorted numbers. In this lecture, we will assume that all the numbers we store are unique.

Operation	Runtime with Unsorted Linked List	Runtime with Sorted Array
Search (i.e. find 17)	$\Theta(n)$	$\Theta(\log n)$ (via binary search)
Select (i.e. find 12 smallest element)	$\Theta(n)$	$\Theta(1)$ (just look at 12th elt.)
Rank (i.e. # elt. less than 17)	$\Theta(n)$	$\Theta(\log n)$
Predecessor/Sucessor[1]	$\Theta(n)$	$\Theta(1)$
Insert element	$\Theta(1)$	$\Theta(n)$
Delete element	$\Theta(1)$[2]	$\Theta(n)$

The above table makes Sorted Arrays look like a pretty decent data structure for *static* data. In many applications, however, the data changes often, in which case the $\Theta(n)$ time it takes to insert or delete an element is prohibitive. This prompts the question: *Is it possible to have one data structure that is the best of both worlds (logarithmic or constant time for all these operations)?* The answer is, essentially, "Yes", provided we don't mind if some of the $\Theta(1)$'s turn into $\Theta(\log n)$. We will sketch out one such data structure, the *Binary Search Tree*.

Before talking about binary search trees, we sketch another data structure called *Heap*. Heaps are worse than both unsorted linked lists and sorted arrays in just about any parameter in the table above, except that they can allow for efficient ($O(\log n)$ time) insertions - still worse than linked lists. But there is one operation that Heaps do exceptionally well (again, $\Theta(\log n)$): $\mathsf{extract-min}$, which outputs the minimum element, and then deletes it from the heap. Heaps (in their simple form that we consider here) still don't beat the asymptotic runtime of binary search trees for any operation, but they have two other advantages: they are much simpler, and if you only want to run insert and $\mathsf{extract-min}$ they are also much faster in practice.

2 Heaps

Definition 1 (Complete binary tree)**.** A complete binary tree is a rooted binary tree where each level is full except maybe the last level, and all nodes on the last level are as far left as they can be.

Definition 2 (Binary min-heap)**.**

A *Binary min-heap* is a data structure that stores elements that have keys from a totally ordered universe (say, the integers). A binary min-heap supports the following operations:

- $\mathsf{insert}(i)$: Inserts an element with key i into the data structure
- $\mathsf{extract-min}$: Returns the element with the minimum key and deletes it from the data structure.

A binary heap stores elements in a complete binary tree with a root r. Each node x has $\mathsf{key}(x)$ (the key of the element stored in x), $\mathsf{p}(x)$ (the parent of x, where $p(r) = \mathsf{NIL}$), $\mathsf{left}(x)$ (the left child of x), and $\mathsf{right}(x)$ (the right child of x). The children of x are either other nodes or NIL.

A binary heap should satisfy two key properties: The first key property is that for every node x, the keys of all nodes under x are greater than $\mathsf{key}(x)$. The second key property of a heap is that its tree is complete (see 1).

As explained above, binary min-heaps (which we call heaps for short) support two operations[3]: $\mathsf{insert}(i)$ and $\mathsf{extract-min}$. These are particularly useful if you want to have a *priority queue* where elements (e.g. jobs) arrive in an arbitrary order, but always leave in order of their key/priority (e.g. when the highest priority job is executed first).

2.1 Basic Operations on Heaps

In the heap data structure we always maintain a pointer to the last node in the heap (the rightmost node in the last level), as well as a pointer to the next node to be created (typically

[3] You can implement other operations such as $\mathsf{search}(i)$ and $\mathsf{delete}(i)$ on a heap, but they would not be efficient (take $\Theta(n)$ time).

the one to the right of the last node; except when the last level is full, in which case this is the left-most node in the next level). Maintaining those is easy when we think of the representation of a heap as an array. But to make things simpler, we will abstract these details and simply assume access to these pointers. (See CLRS for details.)

2.1.1 insert(i)

To insert an element, we create a new node with key i at the bottom of the heap. Then recursively "propagate it up the heap" until the heap property is restored: That is compare i to the key of the new element's parent; if i is smaller than the key of its parent, replace the keys, and continue to compare with the parent's parent, etc. Once the algorithm reaches a parent whose key is smaller, the insertion is complete.

Algorithm 1: Heap insert(i)

```
$x \leftarrow$ first node without two children
$y \leftarrow$ new node with $\mathrm{key}(y) \leftarrow i$, $\mathrm{left}(y) \leftarrow \mathrm{NIL}, \mathrm{right}(y) \leftarrow \mathrm{NIL}$, $\mathrm{p}(y) \leftarrow x$
if $\mathrm{left}(x) == \mathrm{NIL}$ then
    $\mathrm{left}(x) \leftarrow y$
else
    $\mathrm{right}(x) \leftarrow y$
while $\mathrm{p}(y) \neq \mathrm{NIL}$ & $\mathrm{key}(\mathrm{p}(y)) > \mathrm{key}(y)$ do
    $v \leftarrow \mathrm{key}(y)$
    $\mathrm{key}(y) \leftarrow \mathrm{key}(\mathrm{p}(y))$
    $\mathrm{key}(\mathrm{p}(y)) \leftarrow v$
    $y \leftarrow \mathrm{p}(y)$
```

2.1.2 extract − min

To extract-min, we save the key of the root, replace it with the key of the last node, and delete the last node. Then we recursively "propagate the key copied from the last node down the tree". That is compare to the children of the root; if the key of the root is larger than the keys of one of the children, swap the smaller key up, and recurse on the corresponding sub-tree. Once the algorithm reaches children whose keys are larger, the heap property is restored. Then we can return the value we saved at the beginning.

Algorithm 2: Heap extract − min

```
$x \leftarrow$ last node
$r \leftarrow$ root; $m \leftarrow$ key($r$)
key($r$) $\leftarrow$ key($x$)
if right(p($x$)) == NIL then
    left(p(($x$)) $\leftarrow$ NIL
else
    right(p($x$)) $\leftarrow$ NIL
delete node($x$)
while True do
    if left($r$) == NIL then
        break
    else if key(left($r$)) $<$ key($r$) & right($r$) == NIL then
        $v \leftarrow$ key($r$)
        key($r$) $\leftarrow$ key(left($r$))
        key(left($r$)) $\leftarrow v$
        $r \leftarrow$ left($r$)
    else if key(left($r$)) $<$ key($r$), key(right($r$)) then
        $v \leftarrow$ key($r$)
        key($r$) $\leftarrow$ key(left($r$))
        key(left($r$)) $\leftarrow v$
        $r \leftarrow$ left($r$)
    else if key(right($r$)) $<$ key($r$), key(left($r$)) then
        $v \leftarrow$ key($r$)
        key($r$) $\leftarrow$ key(right($r$))
        key(right($r$)) $\leftarrow v$
        $r \leftarrow$ right($r$)
    else
        break
return $m$
```

3 Binary Search Trees

Definition 3 (Binary search tree)**.**

A *binary search tree (BST)* is a data structure that stores elements that have keys from a totally ordered universe (say, the integers). In this lecture, we will assume that each element has a unique key. A BST supports the following operations:

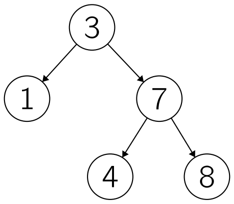
- search(i): Returns an element in the data structure associated with key i
- insert(i): Inserts an element with key i into the data structure
- delete(i): Deletes an element with key i from the data structure, if such an element

exists

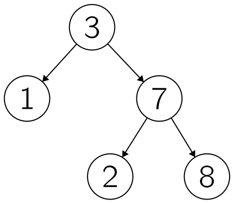
A BST stores the elements in a binary tree with a root r. Each node x has $\mathsf{key}(x)$ (the key of the element stored in x), $\mathsf{p}(x)$ (the parent of x, where $p(r) = \mathsf{NIL}$), $\mathsf{left}(x)$ (the left child of x), and $\mathsf{right}(x)$ (the right child of x). The children of x are either other nodes or NIL.

The key BST property is that for every node x, the keys of all nodes under $\mathsf{left}(x)$ are less than $\mathsf{key}(x)$ and the keys of all nodes under $\mathsf{right}(x)$ are greater than $\mathsf{key}(x)$.

Example 4. In the following example, the root node r stores 3 in it ($\mathsf{key}(r) = 3$), its left child $\mathsf{left}(x)$ stores 1, its right child $\mathsf{right}(x)$ stores 7, and all leaf nodes (storing 1, 4, and 8, respectively) have NIL as their two children.



Example 5. The following binary tree is not a BST since $2 > 3$ and 2 is a child of 7, which is the right child of 3:



Some properties. *Relationship to Quicksort*: We can think of each node x as a pivot for quicksort for the keys in its subtree. The left subtree contains $A_<$ for $\mathsf{key}(x)$ and the right subtree contains $A_>$ for $\mathsf{key}(x)$.

Sorting the keys: We can do an *inorder* traversal of the tree to recover the nodes in sorted order from left to right (the smallest element is in the leftmost node and the largest element is in the rightmost node). The Inorder procedure takes a node x and returns the keys in the subtree under x in sorted order. We can recursively define Inorder(x) as: (1) If left(x) $\neq$ NIL, then run Inorder(left(x)), then: (2) Output key(x) and then: (3) If right(x) $\neq$ NIL, run Inorder(right(x)). With this approach, for every x, all keys in its left subtree will be output before x, then x will be output and then every element in its right subtree.

Subtree property: If we have a subtree where x has y as a left child, y has z as a right child, and z is the root for the subtree T_z, then our BST property implies that all keys in T_z are $> y$ and $< x$. Similarly, if we have a subtree where x has y as a right child, y has z as a left child, and z is the root of the subtree T_z, then our BST property implies that all keys in T_z are between x and y.

3.1 Basic Operations on BSTs

The three core operations on a BST are search, insert, and delete. For this lecture, we will assume that the BST stores distinct numbers, i.e. we will identify the objects with their names and we will have each name be represented by a number.

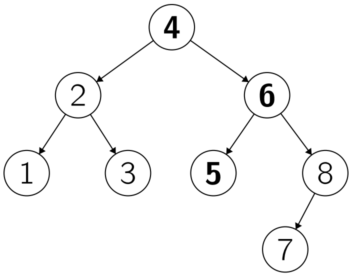
3.1.1 search

To search for an element, we start at the root and compare the key of the node we are looking at to the element we are searching for. If the node's key matches, then we are done. If not, we recursively search in the left or right subtree of our node depending on whether this node was too large or too small, respectively. If we ever reach NIL, we know the element does not exist in our BST. In the following algorithm in this case, we simply return the node that would be the parent of this node if we inserted it into our tree.

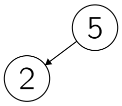
```
Algorithm 3: search(i)
return search(root, i)
```

```
Algorithm 4: search(x, i)
if key(x) == i then
    return x
else if i < key(x) then
    if left(x) == NIL then
        return x
    else
        return search(left(x), i)
else if i > key(x) then
    if right(x) == NIL then
        return x
    else
        return search(right(x), i)
```

Example 6. If we call search(5.5) on the following BST, we will return the node storing 5 (by taking the path in bold). This also corresponds to the path taken when calling search(4.5) or search(5).



Claim 7. search(i) *returns a node containing i if $i \in$ BST, otherwise a node x such that either* key(x) *is the smallest in* BST $> i$ *or the largest in* BST $< i$, *where node x happens to be the parent of the new node if it were to be inserted into* BST. *This follows directly from the BST property. Try to formally prove this claim as an exercise.*



Remark 8. search(i) does **not** necessarily return the node in BST that is closest in value. Consider the tree above. If we search for an element with a key of 4, the node with key 2 is returned, whereas the element with key 5 is the closest element by value.

3.1.2 insert

As before, we will assume that all keys are distinct. We will search(i) for a node x to be the parent and create a new node y, placing it as a child of x where it would logically go according to the BST property.

Algorithm 5: insert(i)

```
$x \leftarrow$ search($i$)
$y \leftarrow$ new node with key($y$) $\leftarrow i$, left($y$) $\leftarrow$ NIL, right($y$) $\leftarrow$ NIL, p($y$) $\leftarrow x$
if $i <$ key($x$) then
    left($x$) $\leftarrow y$
else
    right($x$) $\leftarrow y$
```

Remark 9. Notice that x needed to have NIL as a child where we want to put y by the properties of our search algorithm.

3.1.3 delete

Deletion is a bit more complicated. To delete a node x that exists in our tree, we consider several cases:

1. If x has no children, we simply remove it by modifying its parent to replace x with NIL.
2. If x has only one child c, either left or right, then we elevate c to take x's position in the tree by modifying the appropriate pointer of x's parent to replace x with c, and also fixing c's parent pointer to be x's parent.
3. If x has two children, a left child c_1 and right child c_2, then we find x's immediate successor z and have z take x's position in the tree. Notice that z is in the subtree under x's right child c_2 and we can find it by running $z \leftarrow$ search(c_2, key(x)). Note that since z is x's successor, it doesn't have a left child, but it might have a right child. If z has a right child, then we make z's parent point to that child instead of z (also fixing

the child's parent pointer). Then we replace x with z, fixing up all relevant pointers: the rest of x's original right subtree becomes z's new right subtree, and x's left subtree becomes z's new left subtree.

(Note that alternatively, we could have used x's immediate predecessor y and followed the same analysis in a mirrored fashion.)

In the following algorithm, if p is the parent of x, child(p) refers to left(p) if x was the left child of p and to right(p) otherwise.

Algorithm 6: delete(i)

```
$x \leftarrow$ search($i$)
if key($x$) $\neq i$ then return
if NIL = left($x$) and NIL = right($x$) then
    child(p($x$)) $\leftarrow$ NIL
    delete-node($x$)
if NIL = left($x$) then
    $y \leftarrow$ right($x$)
    p($y$) $\leftarrow$ p($x$)
    child(p($y$)) $\leftarrow y$
    delete-node($x$)
else if NIL = right($x$) then
    $y \leftarrow$ left($x$)
    p($y$) $\leftarrow$ p($x$)
    child(p($y$)) $\leftarrow y$
    delete-node($x$)
else $x$ has two children
    $z \leftarrow$ search(right($x$), key($x$))
    $z' \leftarrow$ right($z$)
    left(p($z$)) $\leftarrow z'$
    p($z'$) $\leftarrow$ p($z$)
    replace $x$ with $z$
    delete-node($x$)
```

3.1.4 Runtimes

The worst-case runtime for search is O(height of tree). As both insert and delete call search a constant number of times (once or twice) and otherwise perform $O(1)$ work on top of that, their runtimes are also O(height of tree).

In the best case, the height of the tree is $O(\log n)$, e.g., when the tree is completely balanced. However, in the worst case it can be $O(n)$ (a long rightward path, for example). This could happen because insert can increase the height of the tree by 1 every time it is called. Currently our operations do not guarantee logarithmic runtimes. To get $O(\log n)$ height we would need to rebalance our tree. There are many examples of self-balancing BSTs, including AVL trees, red-black trees, splay trees (somewhat different but super cool!), etc. Today, we will talk about red-black trees.

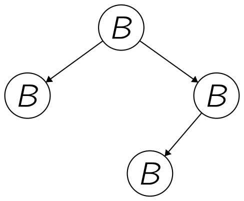
4 Red-Black Trees

One of the most popular balanced BST is the red-black tree developed by Guibas and Sedgewick in 1978. In a red-black tree, all leaves are assumed to have NILs as children.

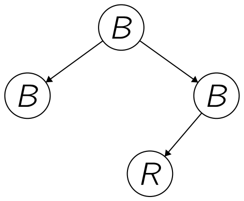
Definition 10. A *red-black tree* is a BST with the following additional properties:

1. Every node is red or black
2. The root is black
3. NILs are black
4. The children of a red node are black
5. For every node x, all x to NIL paths have the same number of black nodes on them

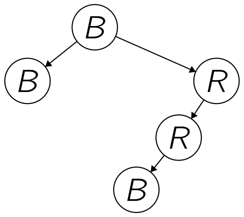
Example 11. (B means a node is black, R means a node is red.) The following tree is *not* a red-black tree since property 5 is not satisfied:



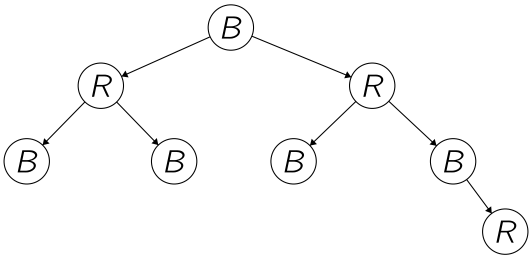
Example 12. (B means a node is black, R means a node is red.) The following tree *is* a red-black tree since all the properties are satisfied:



Example 13. (B means a node is black, R means a node is red.) The following tree is *not* a red-black tree since property 4 is not satisfied:



Example 14. (B means a node is black, R means a node is red.) The following tree *is* a red-black tree:



Remark 15. Intuitively, red nodes represent when a path is becoming too long.

Claim 16. *Any valid red-black tree on n nodes (non-*NIL*) has height $\leq 2\log_2(n+1) = O(\log n)$.*

Proof. For some node x, let $b(x)$ be the "black height" of x, which is the number of black nodes on a $x \to$ NIL path excluding x. We first show that the number of non-NIL descendants of x is at least $2^{b(x)} - 1$ (including x) via induction on the height of x.

Base case: NIL node has $b(x) = 0$ and $2^0 - 1 = 0$ non-NIL descendants. ✓

For our inductive step, let $d(x)$ be the number of non-NIL descendants of x. Then

$$\begin{aligned} d(x) &= 1 + d(\text{left}(x)) + d(\text{right}(x)) \\ &\geq 1 + (2^{b(x)-1} - 1) + (2^{b(x)-1} - 1) \text{ (by induction)} \\ &= 2^{b(x)} - 1 \checkmark \end{aligned}$$

Notice that $b(x) \geq \frac{h(x)}{2}$ (where $h(x)$ is the height of x) since on any root to NIL path there are no two consecutive red nodes, so the number of black nodes is at least the number of red nodes, and hence the black height is at least half of the height. We apply this and the above inequality to the root r (letting $h = h(r)$) to obtain $n \geq 2^{b(r)} - 1 \geq 2^{\frac{h}{2}} - 1$, and hence $h \leq 2\log(n+1)$. □

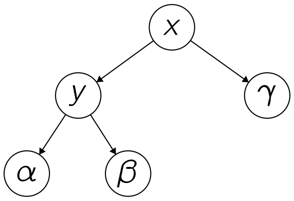
Here is some intuition on why the tree is roughly balanced.

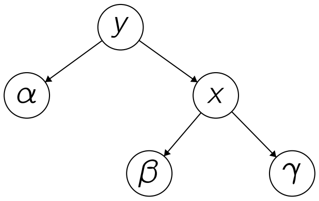
Intuition: By Property (5) of a red-black tree, all $r \to NIL$ paths have $b(r)$ black nodes (excluding the root). Therefore, all these paths have length $\geq b(r)$. However, they also have length $\leq 2 \cdot b(r)$: by Property (4), the number of red nodes is limited to half of the path, since every red node must be followed by a black node, and hence the number of black nodes is at least half of the length of the path. Hence, the lengths of all paths from r to a NIL are within a factor of 2 of each other, and the tree must be reasonably balanced.

Today we will take a brief look at how the red-black tree properties are maintained. Our coverage here is detailed, but not comprehensive, and meant as a case study. For complete coverage, please refer to Chapter 13 of CLRS.

4.1 Rotations

Red-black trees, as do other balanced BSTs, use a concept called rotation. A tree rotation restructures the tree shape locally, usually for the purpose of balancing the tree better. A rotation preserves the BST property (as shown in the following two diagrams). Notably, tree rotations can be performed in $O(1)$ time.





Moving from the first tree to the second is known as a *right rotation of* x. The other direction (from the second tree to the first) is a *left rotation of* y. Notice that we only move the β subtree, which is why we preserve the BST property.

4.2 Insertion in a Red-Black Tree

Let's see how we can perform Insert(i) on a red-black tree while still maintaining all of its properties. The process for inserting a new node is initially similar to that of insertion into any BST.

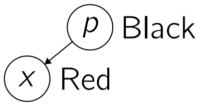
Algorithm 7: insert_rb(i)

```
$p \leftarrow$ search($i$)
$x \leftarrow$ new node with key($x$) $\leftarrow i$, left($x$) $\leftarrow$ NIL, right($x$) $\leftarrow$ NIL, p($x$) $\leftarrow p$
if $i <$ key($p$) then
    left($p$) $\leftarrow x$
else
    right($p$) $\leftarrow x$
color($x$) $\leftarrow$ red
recolor if needed
```

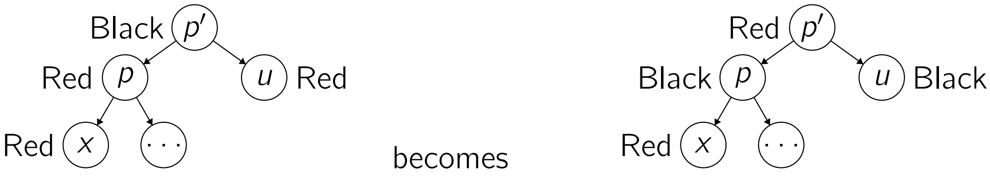
Note that when x is inserted as a red node: Property (1) is satisfied, as we colored the new node red; Property (2) is satisfied, as we did not touch the root; Property (3) is satisfied, as we can color the new NILs black; and Property (5) is satisfied, as we did not change the number of black nodes in the tree. Thus, the only invariant we have to worry about is Property (4), that red nodes have black children.

The recoloring step is broken down into multiple cases. We consider each of them:

Case 1: p is black. In this case, Property (4) is also maintained. So, we simply add x as a new red child of p, and the red-black tree properties are maintained.



Case 2: p is red, and x's uncle u is red. In this case, we insert a red x, change p and u to black, and change p' to red. Because we switched the colors of two nodes on each of these paths (one red→black and one black→red), the number of black nodes on each path is unchanged, so Property (5) remains unchanged. If the parent of p', p'', is black, then Property (4) is maintained. Otherwise, if p'' is red and breaks Property (4) by introducing a "double-red" pair of nodes (p' and p''), then we have to recolor recursively starting at p'.

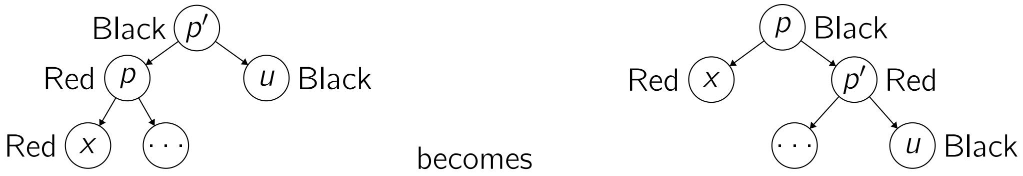


Case 3: p is red, and u is black. There are two possibilities here:

1. We are inserting x as a leaf node. Then, u must be NIL for the red-black tree to have been valid before inserting x. We insert a red x.

2. We are not inserting x; rather, we are recoloring the tree at x from the recursive call in Case 2. We aim to recolor the tree and maintain the number of black nodes on each path from the root to NIL. Note that in this case, x actually has nodes under it.

In both cases, x is red, so we make p black, make p' red, and do a right rotation at p'. We can see that this also maintains the same number of black nodes on each path from the root to NIL, and satisfies Property (4) below p because the original tree was a red-black tree.



If we end up in Case 2 and recursively call recolor, then in the worst case the recursion will bottom out when we hit the root, with a constant number of relabelings and rotations at each level. So, it will be an $O(h)$ operation overall, where h is the height of the tree.

In the analysis above, we considered the cases where x is a left child of p and u is a right child of its parent p'. These cases are representative, showing most of the machinery that we'll need to insert an arbitrary element into an arbitrary red-black tree. (Within Case 2 and Case 3, there are actually a total of four cases each, where p's tree and p''s children could each be swapped, but the recoloring procedure is similar. You are encouraged to read the text for details.)

To summarize, the following is the algorithm for recoloring, in the case where x is a left child and u is a right child.

Algorithm 8: recolor(x)   `// x is a left child, u is a right child`

```
$p \leftarrow$ parent($x$)
if $black$ = color($p$) then
    return
$p' \leftarrow$ parent($p$)
$u \leftarrow$ right($p'$)
if $red$ = color($u$) then
    color($p$) $\leftarrow$ black
    color($u$) $\leftarrow$ black
    color($p'$) $\leftarrow$ red
    recolor($p'$)
else if $black$ = color($u$) then
    color($p$) $\leftarrow$ black
    color($p'$) $\leftarrow$ red
    right_rotate($p'$)
```

Based on our analysis above, we can update our red-black trees in $O(h)$ time upon insertion, where h is the height of the tree. The other operations are similar, and also give the guarantee of worst-case performance of $O(h)$ search, insertion, and deletion. Together with Claim 2, which states that $h = O(\log n)$, we get:

Claim 17. *Red-black trees support* insert, delete, *and* search *in* $O(\log n)$ *time.*

As we have seen, BSTs are very nice – they allow us to maintain a set and report membership, insert, and delete in $O(\log n)$ time. In addition to these basic underlying operations, we can also support other types of queries efficiently. Because the elements are stored maintaining the binary search tree property, we can search for the next largest element or the elements on a range very efficiently. But what if we don't care about these properties? What if we only need to support membership queries? Can we improve our performance of $O(\log n)$ time to nearly constant time? This question motivates our discussion of hash tables, which we will cover in the next lecture.

CS 161 (Stanford, Winter 2022) Lecture 8

Adapted From Virginia Williams' lecture notes. Additional credits go to Luke Johnson, Moses Charikar, Gregory Valiant.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

Hashing

1 Hash tables

A hash table is a commonly used data structure to store an unordered set of items, allowing constant time inserts, lookups and deletes (in expectation). Every item consists of a unique identifier called a *key* and a piece of information. For example, the key might be a Social Security Number, a driver's license number, or an employee ID number. The way in which a hash table stores a item depends only on its key, so we will only focus on the key here, but keep in mind that each key is usually associated with additional information that is also stored in the hash table.

A hash table supports the following operations:

- Insert(k): Insert key k into the hash table.
- Lookup(k): Check if key k is present in the table.
- Delete(k): Delete the key k from the table.

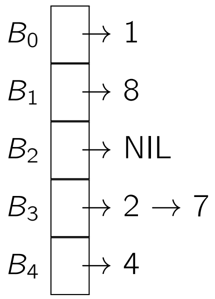
Each operation will take constant time (in expectation).

1.1 Implementation

Let U be the universe of all keys. For example, U could be the set of all 64 bit strings. In this case $|U| = 2^{64}$. This is a very large universe, but we do not need to store all of these 2^{64} keys, we only need to store a subset $S \subset U$. Suppose that we know that the size of the subset we will need to store is less than or equal to n, which is much less than the size of the universe $|U|$. In a hash table of size n, each key $k \in U$ is mapped to one of n "buckets" by a hash function $h : U \to \{1, 2, \ldots, n\}$. Since the universe U is much larger than n, multiple keys could map to the same hash bucket. To accommodate this, each bucket contains a linked list of keys currently stored in that bucket.

Example

Suppose we have a hash table of size $n = 5$ with hash function $h(x) = 13x + 2 \mod 5$. After inserting the elements $\{1, 2, 4, 7, 8\}$ the hash table looks like this:



Where arrows denote pointers in the linked lists, and B_2 is empty. For example, 1 is placed into bucket B_0 because $h(1) = 15 \mod 5 = 0$.

Time complexity

With this setup, the time required to perform an Insert, Lookup, or Delete operation on key k is linear in the length of the linked list for the bucket that key k maps to. We just use the hash function to find the correct bucket for an input key k, and then search the corresponding linked list for the element, inserting or deleting if necessary. Note that an Insert could be performed in constant time by always inserting at the head of the list, but we first need to check if key k is already present.

Choice of size of hash table

The hash table size is usually chosen so that the size of the hash table is at least as large as the maximum number of keys we will need to store at any point of time. If this condition is violated and the number of keys stored grows much larger than the size of the hash table, an implementation will usually increase the size of the table, and recompute the new table from scratch by mapping all keys to the bigger table. Our analysis ignores these complications and assumes that the number of keys is at most the hash table size.

Potential problem with this implementation

In order for the operations to be implemented efficiently, we would like the keys to be distributed uniformly amongst the buckets in the hash table. We might hope that all buckets have at most a constant number of keys mapped to them, so that all operations could be performed in constant time. But for any fixed choice hash function h, one can always produce a subset of keys S such that all keys in S are mapped to the same location in the hash table. In this case, the running times of all operations will be linear in the number of keys – far from the constant we were hoping for. Thus, for a fixed hash function h, it is impossible to give worst case guarantees for the running times of hash table operations.

Possible solutions

There are two styles of analysis that we could use to circumvent this problem:

1. Assume that the set of keys stored in the hash table is random, or

2. Assume that the hash function h is random.

Both are plausible alternatives. The problem with the first alternative is that it is hard to justify that the set of keys stored in the hash table is truly random. It would be more satisfying to have an analysis that works for any subset of keys currently in the hash table. In these notes, we will explore the second alternative, i.e., assume that the hash function h is random.

1.2 Hashing with a completely random hash function

What does it mean for h to be random? One possibility is that h is chosen uniformly and at random from amongst the set of all hash functions $h : U \to \{1, 2, \ldots, n\}$. In fact picking such a hash function is not really practical. Note that there are $n^{|U|}$ possible hash functions. Representing just one of these hash functions requires $\log\left(n^{|U|}\right) = |U| \cdot \log n$ bits. In fact, this means we need to write down $h(x)$ for every $x \in U$ in order to represent h. That's a lot of storage space! Much more than the size of the set we are trying to store in the hash table. One could optimize this somewhat by only recording $h(x)$ for all keys x seen so far (and generating $h(x)$ randomly on the fly when a new x is encountered), but this is impractical too. How would we check if a particular key x has already been encountered? Looks like we would need a hash table for that. But wait, isn't that what we set out to implement? Overall, it is clear that picking a completely random hash function is completely impractical.

Despite this, we will analyze hashing assuming that we have a completely random hash function and then explain how this assumption can be replaced by something that is practical.

Expected cost of hash table operations with random hash function

What is the expected cost of performing any of the operations Insert, Lookup, or Delete with a random hash function? Suppose that the keys currently in the hash table are $x_1, \ldots, x_n$. Consider an operation involving key x_i. The cost of the operation is linear in the size of the hash bucket that x_i maps to. Let X be the size of the hash bucket that x_i maps to. X is a random variable and

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{j=1}^{n} \mathbb{P}[h(x_i) = h(x_j)] \\
&= 1 + \sum_{j \neq i} \mathbb{P}[h(x_i) = h(x_j)] \\
&= 1 + \frac{n-1}{n} \leq 2.
\end{aligned}$$

Here the last step follows from the fact that $\mathbb{P}[h(x_i) = h(x_j)] = 1/n$ when h is random. Note that each key appears in the hash table at most once.

Thus the expected cost of any hashing operation is a constant.

1.3 Universal hash functions

Can we retain the expected cost guarantee of the previous section with a much simpler (i.e., practical) family of hash functions? In the analysis of the previous section, the only fact we used about random hash functions was that $\mathbb{P}[h(x_i) = h(x_j)] = 1/n$. Is it possible to construct a small, practical subset of hash functions with this property?

Thinking along these lines, in 1978, Carter and Wegman introduced the notion of *universal hashing*: Consider a family F of hash functions from U to $\{1, 2, \ldots, n\}$. We say that F is universal if, for every $x_i \neq x_j$, for an h chosen randomly from F, $\mathbb{P}[h(x_i) = h(x_j)] \leq 1/n$.

Clearly the analysis of the previous section shows that for any universal family, the constant expected running time guarantee applies. The family of all hash functions is universal. Is there a simpler universal family?

1.4 A universal family of hash functions

Suppose that the elements of the U are encoded as non-negative integers in the range $\{0, \ldots, |U| - 1\}$. Pick a prime $p \geq |U|$. For $a, b \in \{0, \ldots p - 1\}$, consider the family of hash functions

$$h_{a,b}(x) = (ax + b \mod p) \mod n.$$

where $a \in \{1, \ldots, p - 1\}$ and $b \in \{0, 1, \ldots, p - 1\}$.

Proposition 1. *This family of hash functions F is universal.*

In order to prove this statement, first, let's count the number of hash functions in this family F. We have $p-1$ choices for a, and p choices for b, so $|F| = p(p-1)$. In order to prove that F is universal, we need to show that for an h chosen randomly from F, $\mathbb{P}[h(x_i) = h(x_j)] \leq 1/n$. Since there are $p(p-1)$ hash functions in F, this is equivalent to showing that the number of hash functions in F that map x_i and x_j to the same output is less than or equal to $\frac{p(p-1)}{n}$. To show that this is true, first consider how $h_{a,b}$ behaves without the $\mod n$. Call these functions $f_{a,b}$:

$$f_{a,b}(x) = ax + b \mod p$$

The $f_{a,b}$ have the following useful property:

Proposition 2. *For a given $x_1, x_2, y_1, y_2 \in \{0, \ldots, p-1\}$ such that $x_1 \neq x_2$ there exists only one function $f_{a,b}$ such that*

$$\begin{aligned} f_{a,b}(x_1) &= y_1, \text{ and} \\ f_{a,b}(x_2) &= y_2 \end{aligned}$$

Proof. Solve the above two equations for a and b:

$$\begin{aligned} ax_1 + b &\equiv y_1 \pmod p \\ ax_2 + b &\equiv y_2 \pmod p \end{aligned}$$

By subtracting the two equations, we get:

$$a(x_1 - x_2) \equiv y_1 - y_2 \pmod p$$

Since p is prime and $x_1 \neq x_2$, the above equation has only one solution for $a \in \{0, \dots, p-1\}$. Then

$$b \equiv y_1 - ax_1 \pmod p$$

So we have found the unique a and b such that $f_{a,b}(x_1) = y_1$ and $f_{a,b}(x_2) = y_2$. □

In the above proof, note that $a = 0$ only when $y_1 = y_2 = b$. This is why we restrict $a \neq 0$, we don't want the hash function mapping all elements to the same value b.

Now, we have shown that for a given x_1, x_2, for each selection of y_1, y_2 with $y_1 \neq y_2$, there is exactly one function $f_{a,b}$ that maps x_1 to y_1 and x_2 to y_2. So, in order to find out how many functions $h_{a,b}$ map x_1 and x_2 to the same value mod n, we just need to count the number of pairs (y_1, y_2) where $y_1 \neq y_2$ and $y_1 \equiv y_2 \pmod n$. There are p possible selections of y_1 for this pair, and then $\leq (p-1)/n$ of the possibilities for y_2 will be equal to y_1 mod n. (Convince yourself that this is true.) This gives a total of $\frac{p(p-1)}{n}$ functions $h_{a,b}$ that map x_1 and x_2 to the same element. So then

$$\begin{aligned}
\mathbb{P}[h_{a,b}(x_1) = h_{a,b}(x_2)] &\leq \frac{p(p-1)/n}{|F|} \\
&= \frac{p(p-1)}{p(p-1)(n)} \\
&= \frac{1}{n}
\end{aligned}$$

which means the family F of the $h_{a,b}$ is universal, as desired.

Wrapping up the discussion on hashing, if we pick a random hash function from this family, then the expected cost of any hashing operation is constant. Note that picking a random hash function from the family simply involves picking a, b – significantly simpler than picking a completely random hash function.

2 Balls and bins

A useful abstraction in thinking about hashing with random hash functions is the following experiment: Throw m balls randomly into n bins. (The connection to hashing should be clear: the balls represent the keys and the bins represent the hash buckets.) The balls into bins experiment arises in several other problems as well, e.g., analysis of load balancing. In the context of hashing, the following questions arise about the balls and bins experiment:

- How large does m have to be so that with probability greater than 1/2, we have (at least) two balls in the same bin? This tells us how large our hash table needs to be to avoid any collisions. We will explore this at the end of these notes.

- Suppose $m = n$; what is the maximum number of balls that fall into a bin? This tells us the size of the largest bucket in the hash table when the number of keys is equal to the number of buckets in the table. We might explore this in the next homework.

No collisions

The first question is related to the so called *birthday paradox*: Suppose you have 23 people in a room. Then (somewhat surprisingly) the probability that there exists some pair with the same birthday is greater than 1/2! (This assumes that birthdays are independent and randomly distributed.) 23 seems like an awfully small number to get a pair with the same birthday. There are 365 days in a year! How do we explain this? Consider throwing m balls into n bins. The expected number of pairs that fall into the same bucket is $m(m-1)/2n$. (This follows from linearity of expectation. Note that the probability that a fixed pair falls into the same bucket is $1/n$.) Thus the probability that there is a collision is upper bounded by the expected number of collisions which is $m(m-1)/2n$. (Convince yourself that this is true.) On the other hand, we can also show that the probability that all m balls fall into distinct bins is at most $e^{-m(m-1)/2n}$:

Proof.

$$\mathbb{P}[\text{no collisions}] = \prod_{i=1}^{m-1}\left(1 - \frac{i}{n}\right)$$

Now, we use the fact that $(1 - x) \leq e^{-x}$:

$$(1 - \frac{i}{n}) \leq e^{-i/n}$$

So

$$\mathbb{P}[\text{no collisions}] \leq \prod_{i=1}^{m-1} e^{-i/n}$$

$$\mathbb{P}[\text{no collisions}] \leq e^{\sum_{i=1}^{m-1} -i/n}$$

$$\mathbb{P}[\text{no collisions}] \leq e^{(-m(m-1)/(2n))}$$

□

For m about $\sqrt{(2\ln 2)n} \approx 1.18\sqrt{n}$ this probability is less than 1/2, i.e., the probability of a collision is greater than 1/2.

This is a useful design principle to keep in mind: If we want to design a hash table with no collisions, then the size of the hash table should be larger than the square of the number of elements we need to store in it. For our purposes in this note, insisting on no collisions means that the number of elements in the hash table can only be a small fraction of the hash table size which is quite wasteful.

The birthday problem calculation is useful in other contexts. Here is an application: Suppose we assign random b-bit IDs to m users. How large does b have to be to ensure that all users have distinct IDs with probability $1 - \delta$. Here $\delta > 0$ is a given error tolerance. Assigning b-bit IDs is identical to mapping to $n = 2^b$ buckets. The birthday problem calculation shows us that the probability of a collision is at most $m^2/2n = m^2/2^{b+1}$. We should set b large enough such that this bound is at most δ. Thus b should be at least $2 \log m - 1 + \log(1/\delta)$.

CS 161 (Stanford, Winter 2022) Lecture 9

Adapted From Virginia Williams' lecture notes. Additional credits go to Romil Veram, Juliana Cook, Seth Hildick-Smith, Gregory Valiant, Mary Wootters.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

Graphs, DFS, and BFS

1 Graphs

A **graph** is a set of **vertices** and **edges** connecting those vertices. Formally, we define a graph G as $G = (V, E)$ where $E \subseteq V \times V$. For ease of analysis, the variables n and m typically stand for the number of vertices and edges, respectively. Graphs can come in two flavors, **directed** or **undirected**. If a graph is undirected, it must satisfy the property that $(i, j) \in E$ iff $(j, i) \in E$ (i.e., all edges are bidirectional). In undirected graphs, $m \leq \frac{n(n-1)}{2}$. In directed graphs, $m \leq n(n-1)$. Thus, $m = O(n^2)$ and $\log m = O(\log n)$. A connected graph is a graph in which for any two nodes u and v there exists a path from u to v. For an undirected connected graph $m \geq n - 1$. A *sparse graph* is a graph with few edges (for example, $\Theta(n)$ edges) while a *dense* graph is a graph with many edges (for example, $m = \Theta(n^2)$).

1.1 Representation

A common issue is the topic of how to represent a graph's edges in memory. There are two standard methods for this task.

An **adjacency matrix** uses an arbitrary ordering of the vertices from 1 to $|V|$. The matrix consists of an $n \times n$ binary matrix such that the (i, j)-th element is 1 if (i, j) is an edge in the graph, 0 otherwise.

An **adjacency list** consists of an array A of $|V|$ lists, such that $A[u]$ contains a linked list of vertices v such that $(u, v) \in E$ (the neighbors of u). In the case of a directed graph, it's also helpful to distinguish between outgoing and ingoing edges by storing two different lists at $A[u]$: a list of v such that $(u, v) \in E$ (the out-neighbors of u) as well as a list of v such that $(v, u) \in E$ (the in-neighbors of u).

What are the tradeoffs between these two methods? To help our analysis, let $\deg(v)$ denote the **degree** of v, or the number of vertices connected to v. In a directed graph, we can distinguish between out-degree and in-degree, which respectively count the number of outgoing and incoming edges.

- The adjacency matrix can check if (i, j) is an edge in G in constant time, whereas the adjacency list representation must iterate through up to $\deg(i)$ list entries.

- The adjacency matrix takes $\Theta(n^2)$ space, whereas the adjacency list takes $\Theta(m+n)$ space.
- The adjacency matrix takes $\Theta(n)$ operations to enumerate the neighbors of a vertex v since it must iterate across an entire row of the matrix. The adjacency list takes $\deg(v)$ time.

What's a good rule of thumb for picking the implementation? One useful property is the sparsity of the graph's edges. If the graph is **sparse**, and the number of edges is considerably less than the max ($m \ll n^2$), then the adjacency list is a good idea. If the graph is **dense** and the number of edges is nearly n^2, then the matrix representation makes sense because it speeds up lookups without too much space overhead. Of course, some applications will have lots of space to spare, making the matrix feasible no matter the structure of the graphs. Other applications may prefer adjacency lists even for dense graphs. Choosing the appropriate structure is a balancing act of requirements and priorities.

2 Depth First Search (DFS)

Given a starting vertex, it's desirable to find all vertices reachable from the start. There are many algorithms to do this, the simplest of which is depth-first search. As the name implies, DFS enumerates the deepest paths, only backtracking when it hits a dead end or an already-explored section of the graph. DFS by itself is fairly simple, so we introduce some augmentations to the basic algorithm.

- To prevent loops, DFS keeps track of a "color" attribute for each vertex. Unvisited vertices are white by default. Vertices that have been visited but still may be backtracked to are colored gray. Vertices which are completely processed are colored black. The algorithm can then prevent loops by skipping non-white vertices.[1]
- Instead of just marking visited vertices, the algorithm also keeps track of the tree generated by the depth-first traversal. It does so by marking the "parent" of each visited vertex, aka the vertex that DFS visited immediately prior to visiting the child.
- The augmented DFS also marks two auto-incrementing timestamps d and f to indicate when a node was first discovered and finished.

The algorithm takes as input a start vertex s and a starting timestamp t, and returns the timestamp at which the algorithm finishes. Let $N(s)$ denote the neighbors of s; for a directed graph, let $N_{\text{out}}(s)$ denote the out-neighbors of s.

[1] In the slides, white, gray, and black are replaced with light green, orange, and dark green, respectively.

Algorithm 1: init(G)

```
foreach $v \in G$ do
    color($v$) $\leftarrow$ white
    d($v$), f($v$) $\leftarrow \infty$
    p($v$) $\leftarrow$ nil
```

Algorithm 2: DFS(s, t): $s \in V$ is white, t = time

```
color($s$) $\leftarrow$ gray
// $d(s)$ is the discovery time of $s$
$d(s) \leftarrow t$
$t \leftarrow t + 1$
// In the loop below, replace $N(s)$ with $N_{\text{out}}(s)$ for directed $G$.
foreach $v \in N(s)$ do
    if color($v$) = white then
        p($v$) $\leftarrow s$
        // Update $t$ to be the finish time of DFS starting at $v$.
        $t \leftarrow$ DFS($v, t$)
        $t \leftarrow t + 1$
// Finish time:
f($s$) $\leftarrow t$
// $s$ is finished.
color($s$) $\leftarrow$ black
return f(s)
```

There are multiple ways we can search using DFS. One way is to search from some source node s, which will give us a set of black nodes reachable from s and white nodes unreachable from s.

Algorithm 3: DFS(s): DFS from a source node s

```
init($G$)
DFS($s$, 1)
```

Another way to use DFS is to search over the entire graph, choosing some white node and finding everything we can reach from that node, and repeating until we have no white nodes remaining. In an undirected graph this will give us all of the connected components.

Algorithm 4: DFS(G): DFS on an entire graph G

```
init($G$)
$t \leftarrow 1$
foreach $v \in G$ do
    if $color(v) = white$ then
        $t \leftarrow$ DFS($v$, $t$)
        $t \leftarrow t + 1$
```

2.1 Runtime of DFS

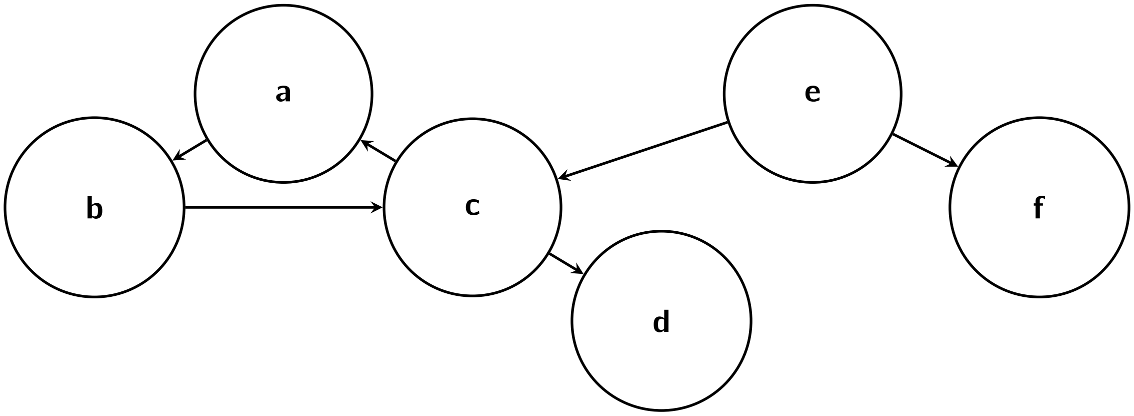
We will now look at the runtime for the standard DFS algorithm (Line 2).

Everything above the loop runs in $O(1)$ time per node visit. Excluding the recursive call, everything inside of the for loop takes $O(1)$ time every time an edge is scanned. Everything after the for loop also runs in $O(1)$ time per node visit.

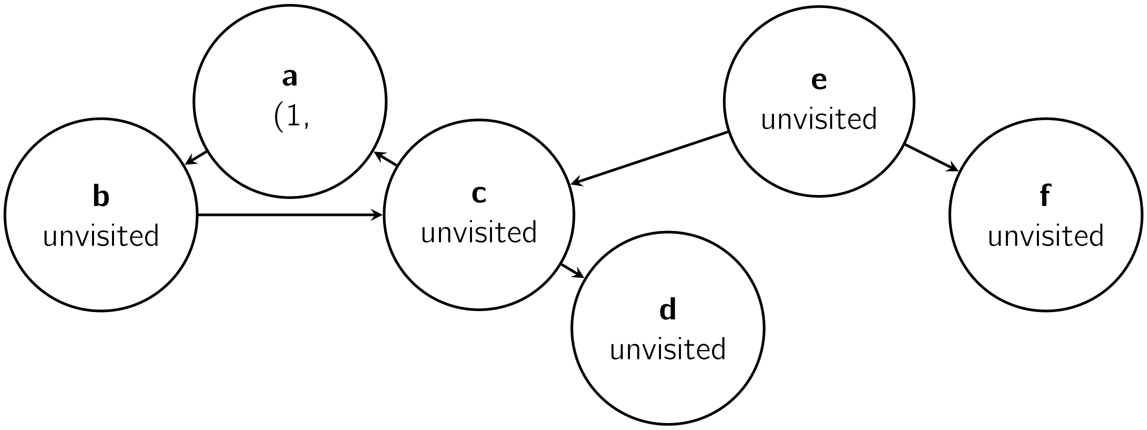
We can express the runtime of DFS as $O(\#\text{ of node visits} + \#\text{ of edge scans})$. Assume we have a graph with n nodes and m edges. We know that the # of node visits is $\leq n$, since we only visit white nodes and whenever we visit a node we change its color from white to gray and never change it back to white again. We also know that an edge (u, v) is scanned only when u or v is visited. Since every node is visited at most once, we know that an edge (u, v) is scanned at most twice (or only once for directed graphs). Thus, # of edges scanned is $O(m)$, and the overall runtime of DFS is $O(m + n)$.

2.2 DFS Example

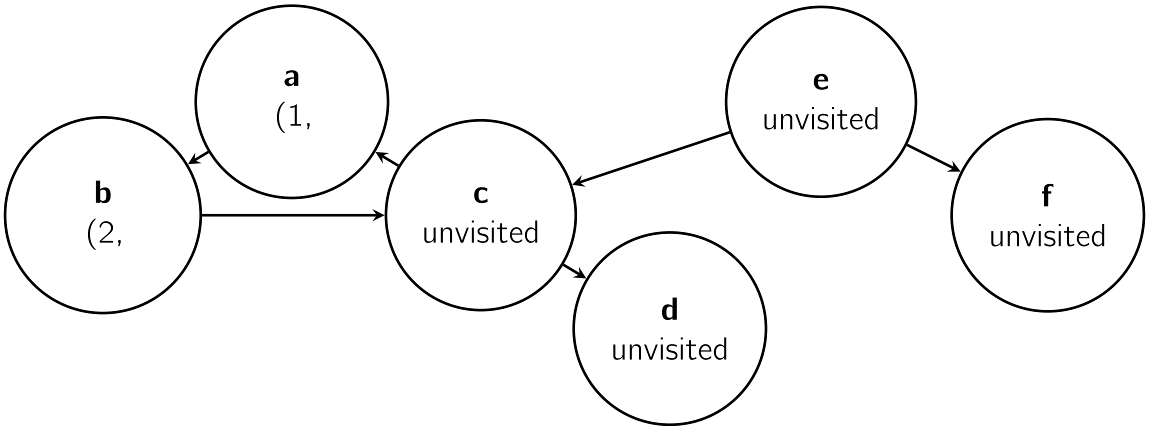
We will now try running DFS on the example graph below.



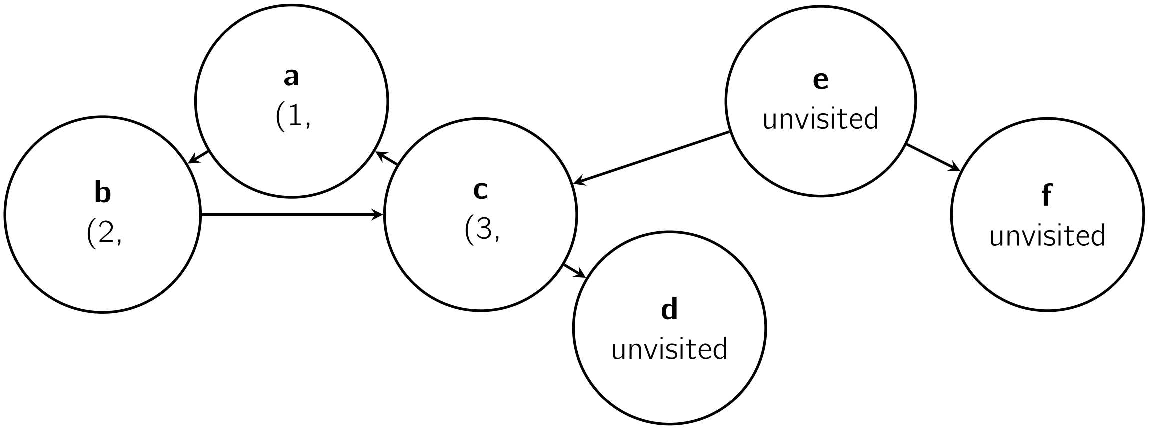
We mark all of the nodes as unvisited and start at a white node, in our case node a.



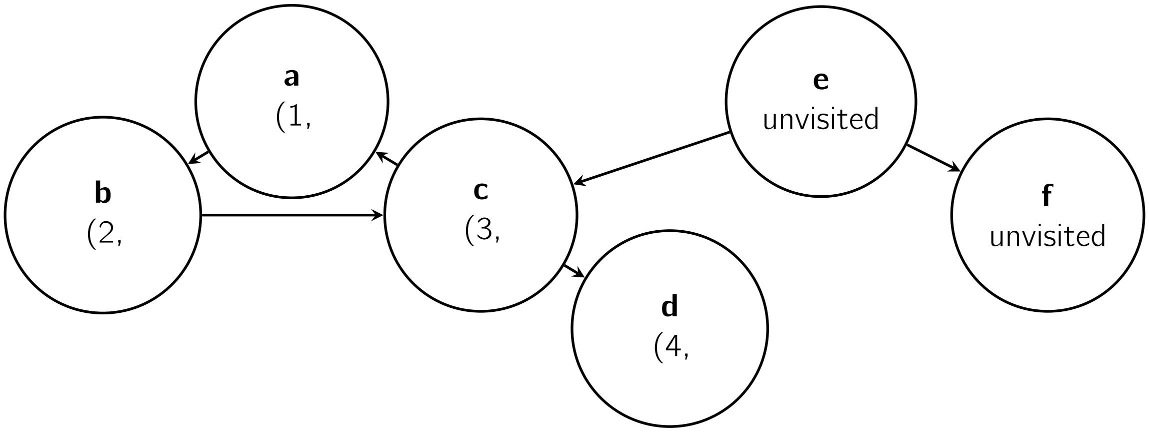
From node a we will visit all of a's children, namely node b.



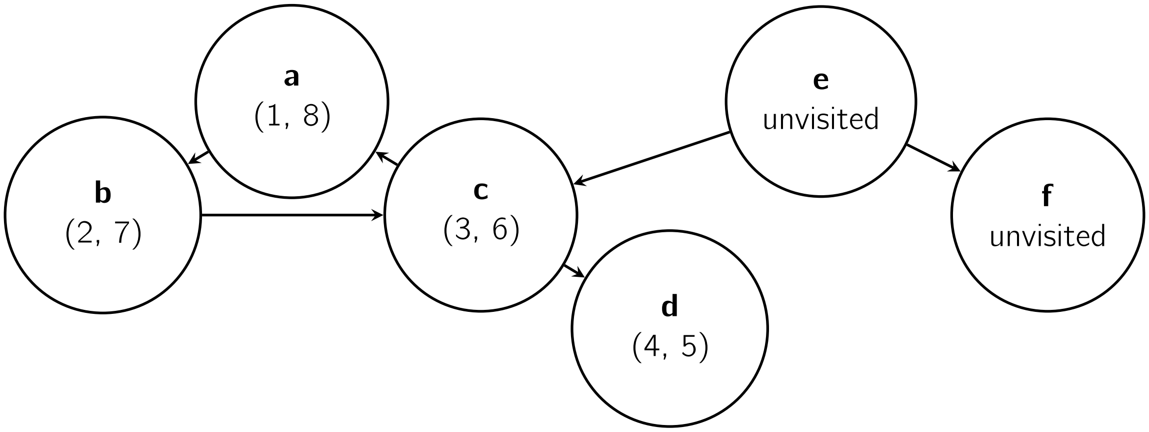
We now visit b's child, node c.



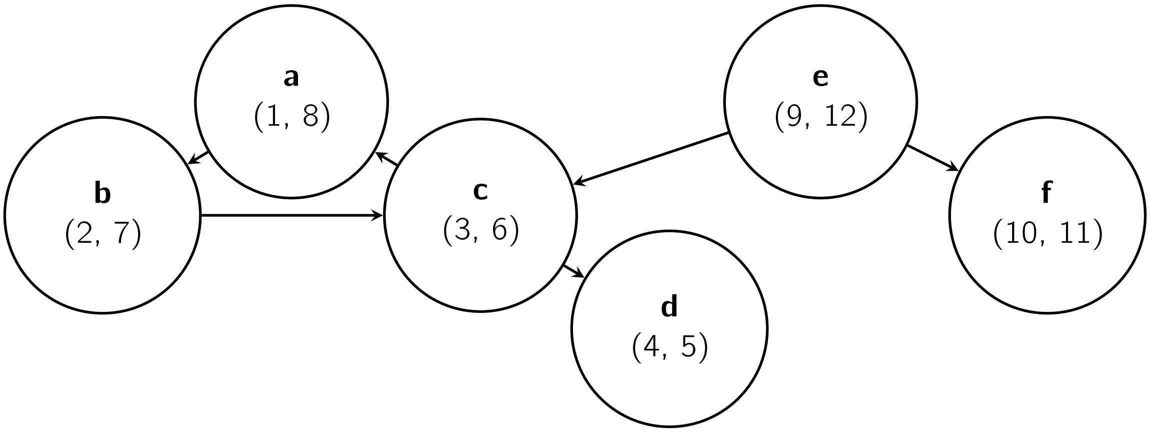
Node c has two children that we must visit. When we try to visit node a we find that node a has already been visited (and would be colored gray, as we are in the process of searching a's children), so we do not continue searching down that path. We will next search c's second child, node d.



Since node d has no children, we return back to its parent node, c, and continue to go back up the path we took, marking nodes with a finish time when we have searched all of their children.



Once we reach our first source node a we find that we have searched all of its children, so we look in the graph to see if there are any unvisited nodes remaining. For our example, we start with a new source node e and run DFS to completion.

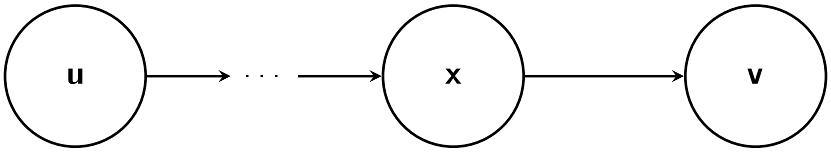


3 Breadth First Search (BFS)

In depth first search, we search “deeper" in the graph whenever possible, exploring edges out of the most recently discovered node that still has unexplored edges leaving it. Breadth first search (BFS) instead expands the frontier between discovered and undiscovered nodes uniformly across the breadth of the frontier, discovering all nodes at a distance k from the source node before nodes at distance $k+1$.

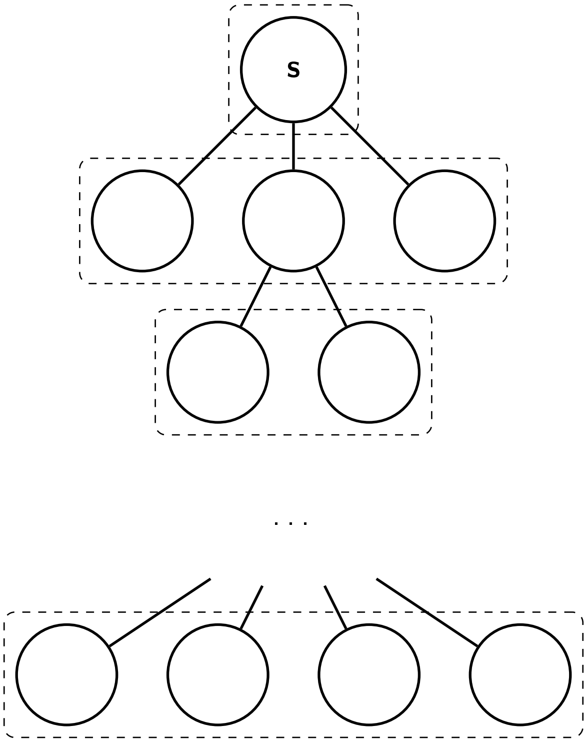
BFS(s) computes for every node $v \in G$ the distance from s to v in G. $d(u,v)$ is the length of the shortest path from u to v.

A simple property of unweighted graphs is as follows: let P be a shortest $u \to v$ path and let x be the node before v on P. Then $d(u,v) = d(u,x) + 1$.



Path P

BFS(s) computes sets L_i, the set of nodes at distance i from s, as seen in the diagram below.



Algorithm 5: BFS(s)

```
Set vis[$v$] ← false for all $v$
Set all $L_i \leftarrow \emptyset$ for $i \in \{1, \dots, n-1\}$
$L_0 \leftarrow \{s\}$
vis[$s$] ← true
for $i = 0, \dots, n-1$ do
    if $L_i = \emptyset$ then
        Exit
    while $L_i \neq \emptyset$ do
        $u \leftarrow L_i$.pop()
        // In the loop below, replace $N$ with $N_{\text{out}}$ for a directed graph.
        foreach $x \in N(u)$ do
            if $vis[x] = false$ then
                vis[$x$] ← true
                $L_{i+1}$.insert($x$)
                $p(x) \leftarrow u$
```

3.1 Runtime Analysis

We will now look at the runtime for our BFS algorithm (Line 5) for a graph with n nodes and m edges. All of the initialization above the first for loop runs in $O(n)$ time. Visiting each node within the while loop takes $O(1)$ time per node visited. Everything inside the inner foreach loop takes $O(1)$ time per edge scanned, which we can simplify to a runtime of $O(m)$ time overall for the entire inner for loop.

Overall, we see that our runtime is $O(\# \text{ nodes visited} + \# \text{ edges scanned}) = O(m+n)$.

3.2 Correctness

We will now show that BFS correctly computes the shortest path between the source node and all other nodes in the graph. Recall that L_i is the set of nodes that BFS calculates to be distance i from the source node.

Proposition 1. *For all i, $L_i = \{x \mid d(s,x) = i\}$.*

Proof. We will prove this by (strong) induction on i.

Base case: $i = 0$, and $L_0 = \{s\}$.

Induction hypothesis: Suppose that $L_j = \{x \mid d(s,x) = j\}$ for every $j \leq i$ (induction hypothesis for i).

Inductive step: We will show two things: (1) if y was added to L_{i+1}, then $d(s,y) = i+1$, and (2) if $d(s,y) = i+1$, then y is added to L_{i+1}. After proving (1) and (2) we can conclude that $L_{i+1} = \{y \mid d(s, y) = i + 1\}$ and complete the induction.

Let's prove (1). First, if y is added to L_{i+1}, it was added by traversing an edge (x, y) where $x \in L_i$, so that there is a path from s to y taking the shortest path from s to x followed by the edge (x, y), and so $d(s, y) \leq d(s, x) + 1$. Since $x \in L_i$, by the induction hypothesis, $d(s, x) = i$, so that $d(s, y) \leq i + 1$. However, since $y \notin L_j$ for any $j \leq i$, by the induction hypothesis, $d(s, y) > i$, and so $d(s, y) = i + 1$.

Let's prove (2). If $d(s, y) = i + 1$, then by the inductive hypothesis $y \notin L_j$ for $j \leq i$. Let x be the node before y on the $s \to y$ shortest path P. As $d(s, y) = i + 1$ and the portion of P from s to x is a shortest path and has length exactly i. Thus, by the induction hypothesis, $x \in L_i$. Thus, when x was scanned, edge (x, y) was scanned as well. If y had not been visited when (x, y) was scanned, then y will be added to L_{i+1}. Hence assume that y was visited before (x, y) was scanned. However, since $y \notin L_j$ for any $j \leq i$, y must have been visited by scanning another edge out of a node from L_i, and hence again y is added to L_{i+1}. □

3.3 BFS versus DFS

If you simplify BFS and DFS to the basics, ignoring all timestamps and levels that we would usually create, BFS and DFS have a very similar structure. Breadth first search explores the nodes closest and then moves outwards, so we can use a queue (first in first out data structure) to put new nodes at the end of the list and pull the oldest/nearest nodes from the top of the list. Depth first search goes as far down a path as it can before coming back to explore other options, so we can use a stack (last in first out data structure) which pushes new nodes on the top and also pulls the newest nodes from the top. See the pseudocode below for more detail.

```
Algorithm 6: DFS($s$): $s$ is the source node
$T$ ← empty stack
Push $s$ onto $T$
while $T$ is not empty do
    $u$ ← pop from top of $T$
    Push all unvisited neighbors of $u$ on top of stack $T$
```

```
Algorithm 7: BFS($s$): $s$ is the source node
$T$ ← empty queue
Push $s$ onto $T$
while $T$ is not empty do
    $u$ ← pop from front of $T$
    Push all unvisited neighbors of $u$ on back of queue $T$
```

CS 161 (Stanford, Winter 2022) Lecture 10

Adapted from Tim Roughgarden's lecture notes. Additional credits go to Luke Johnston and Mary Wootters.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

Strongly Connected Components

1 Connected components in undirected graphs

A **connected component** of an undirected graph $G = (V, E)$ is a maximal set of vertices $S \subset V$ such that for each $u \in S$ and $v \in S$, there exists a path in G from vertex u to vertex v.

Definition 1 (Formal Definition)**.** Let $u \sim v$ if and only if G has a path from vertex u to vertex v. This is an equivalence relation (it is symmetric, reflexive, and transitive). Then, a connected component of G is an equivalence class of this relation $\sim$. Recall that the equivalence class of a vertex u over a relation $\sim$ is the set of all vertices v such that $u \sim v$.

1.1 Algorithm to find connected components in a undirected graph

In order to find a connected component of an undirected graph, we can just pick a vertex and start doing a search (BFS or DFS) from that vertex. All the vertices we can reach from that vertex compose a single connected component. To find all the connected components, then, we just need to go through every vertex, finding their connected components one at a time by searching the graph. Note however that we do not need to search from a vertex v if we have already found it to be part of a previous connected component. Hence, if we keep track of what vertices we have already encountered, we will only need to perform one BFS for each connected component.

Proof. When searching from a particular vertex v, we will clearly never reach any nodes outside the connected component with DFS or BFS. So we just need to prove that we will in fact reach all connected vertices. We can prove this by induction: Consider the vertices at minimum distance i from vertex v. Call these vertices "level i" vertices. If BFS or DFS successfully reaches all vertices at level i, then they must reach all vertices at level $i + 1$, since each vertex at distance $i + 1$ from v must be connected to some vertex at distance i from v. This is the inductive step, and for the base case, DFS or BFS will clearly reach all vertices at level 0 (just v itself). So indeed this algorithm will find each connected component correctly. □

The searches in the above algorithm take total time $O(|E| + |V|)$, because each BFS or DFS call takes linear time in the number of edges and vertices for its component, and each

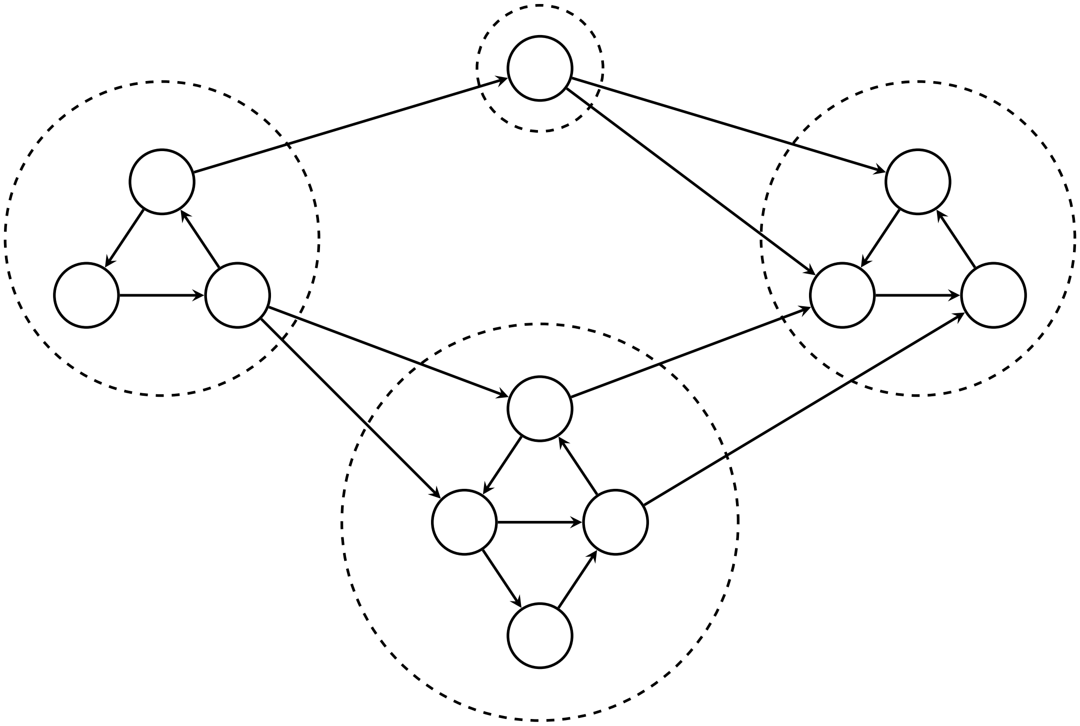


Figure 1: The strongly connected components of a directed graph.

component is only searched once, so all searches will take time linear in the total number of edges and vertices.

2 Connectivity in directed graphs

How can we extend the notion of connected components to directed graphs?

Definition 2 (Strongly connected component (SCC))**.** A strongly connected component in a directed graph $G = (V, E)$ is a maximal set of vertices $S \subset V$ such that each vertex $v \in S$ has a path to each other vertex $u \in S$. This is the same as the definition using equivalence classes for undirected graphs, except now $u \sim v$ if and only if there is a path from u to v AND a path from v to u.

Definition 3 (Weakly connected component)**.** Let $G = (V, E)$ be a directed graph, and let G' be the undirected graph that is formed by replacing each directed edge of G with an undirected edge. Then the weakly connected components of G are exactly the connected components of G'.

3 Algorithm to find strongly connected components of a directed graph

The algorithm we present is essentially two passes of depth-first search, plus some extremely clever additional book-keeping. The algorithm is described in a top-down fashion in Algorithms 1 to 3. Algorithm 1 describes the top level of the algorithm, and Algorithm 2 and Algorithm 3 describe the subroutines DFS-Loop and DFS. Read these procedures carefully before proceeding to the next section.

Algorithm 1: The top level of our SCC algorithm. The f-values and leaders are computed in the first and second calls to DFS-Loop, respectively (see below).

```
Input : A directed graph $G = (V, E)$, in adjacency list representation. Assume that the
        vertices $V$ are labeled $1, 2, 3, \dots, n$.
$G^{\text{rev}} \leftarrow$ the graph $G$ after the orientation of all arcs have been reversed.
Run the DFS-Loop subroutine on $G^{\text{rev}}$, processing vertices in any arbitrary order, to
  obtain a finishing time $f(v)$ for each vertex $v \in V$.
Run the DFS-Loop subroutine on $G$, processing vertices in decreasing order of $f(v)$, to
  assign a "leader" to each vertex $v \in V$. The leader of a vertex $v$ will be the source
  vertex that the DFS that discovered $v$ started from.
The strongly connected components of $G$ correspond to vertices of $G$ that share a
  common leader.
```

Remark 4. The algorithm in Algorithm 1 is a bit different than the one in CLRS/Lecture! The difference is that in these notes, we first run DFS on the reversed graph, and then we run it again on the original; in CLRS, we first run DFS on the original, and then the second time on the reversed graph. Is it the case that one of these two textbooks has messed it up? In fact, it doesn't matter: the SCCs of G are the same as the SCCs of G^{rev}, so both algorithms find exactly the same SCC decomposition.

As we've seen, each invocation of DFS-Loop can be implemented in linear time (i.e., $O(|E| + |V|)$), so this whole algorithm will take linear time (the bookkeeping of leaders and finishing times just adds a constant number of operations per each node).

4 An Example

But why on earth should this algorithm work? An example should increase its plausibility (though it certainly doesn't constitute a proof of correctness). Figure 2 displays a reversed graph G^{rev}, with its vertices numbered arbitrarily, and the f-values computed in the first call to DFS-Loop. In more detail, the first DFS is initiated at node 9. The search must proceed next to node 6. DFS then has to make a choice between two different adjacent nodes; we have shown the f-values that ensue when DFS visits node 3 before node 8.[1] When DFS visits

[1] Different choices of which node to visit next generate different sets of f-values, but our proof of correctness will apply to all ways of resolving these choices.

Algorithm 2: The DFS-Loop subroutine.

```
Input: A directed graph $G = (V, E)$, in adjacency list representation.
Let global variable $t \leftarrow 0$. /* This keeps track of the number of vertices that
    have been fully explored.                                                      */
Let global variable $s \leftarrow$ NULL. /* This keeps track of the vertex from which
    the last DFS call was invoked.                                                 */
for $i = n, n-1, \ldots, 1$ do
    // In the first call, vertices are labeled $1, 2, \ldots, n$ arbitrarily.  In
       the second call, vertices are labeled by their $f(v)$-values from
       the first call.
    if $i$ not yet explored then
        Let $s \leftarrow i$. /* Set the current source $s$ to $i$.  All vertices
            discovered from the below DFS call will have their leader set
            to $s$.                                                                */
        DFS($G, i$)
```

Algorithm 3: The DFS subroutine. The f-values only need to be computed during the first call to DFS-Loop, and the leader values only need to be computed during the second call to DFS-Loop.

```
Input: A directed graph $G = (V, E)$, in adjacency list representation, and a source
       vertex $i \in V$.
Mark $i$ as explored. /* It remains explored for the entire duration of the
    DFS-Loop call.                                                                 */
leader($i$) $\leftarrow s$
foreach arc $(i, j)$ in $G$ do
    if $j$ not yet explored then
        DFS($G, j$)
$t \leftarrow t + 1$
Let $f(i) \leftarrow t$
```

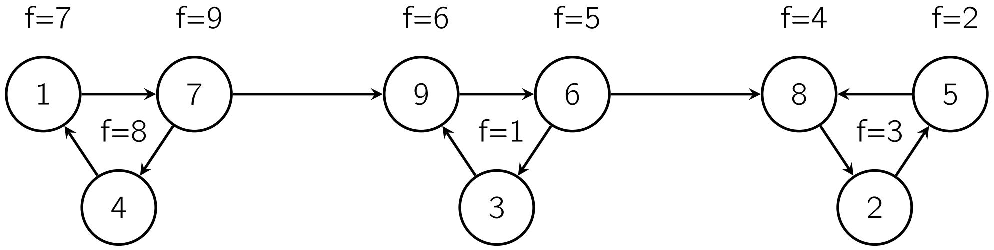


Figure 2: Example execution of the strongly connected components algorithm. Nodes are labeled arbitrarily and their finishing times are shown.

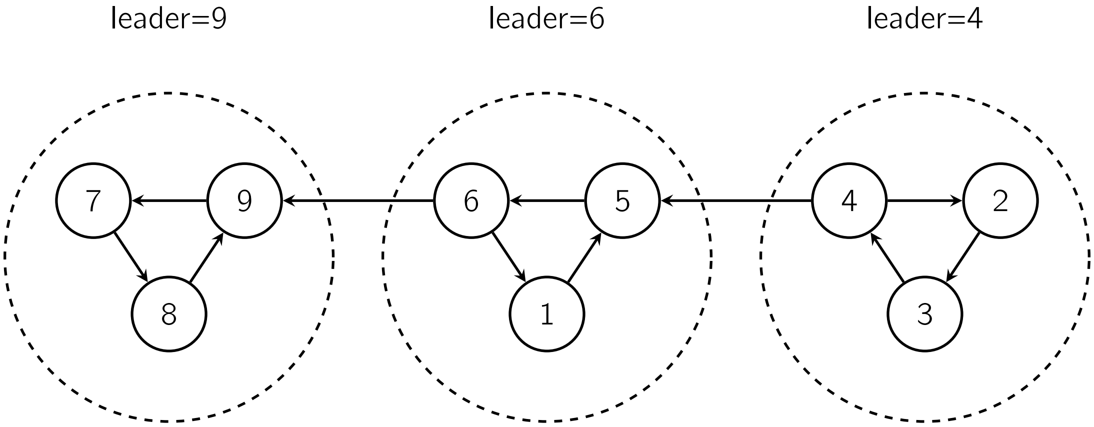


Figure 3: Example execution of the strongly connected components algorithm. Nodes are labeled by their finishing times and their leaders are shown.

node 3 it gets stuck; at this point node 3 is assigned a finishing time of 1. DFS backtracks to node 6, proceeds to node 8, then node 2, and then node 5. DFS then backtracks all the way back to node 9, resulting in nodes 5, 2, 8, 6, and 9 receiving the finishing times 2, 3, 4, 5, and 6, respectively. Execution returns to DFS-Loop, and the next (and final) call to DFS begins at node 7.

Figure 3 shows the original graph (with all arcs now unreversed), with nodes labeled with their finishing times. The magic of the algorithm is now evident, as the SCCs of G present themselves to us in order: since we call DFS on the nodes in decreasing order of their finishing times, the first call to DFS discovers the nodes 7–9 (with leader 9); the second the nodes 1, 5, and 6 (with leader 6); and the third the remaining three nodes (with leader 4).

4.1 The Acyclic Meta-Graph of SCCs

First, observe that the strongly connected components of a directed graph form an acyclic "meta-graph", where the meta-nodes correspond to the SCCs $C_1, \ldots, C_k$, and there is an arc $C_h \to C_\ell$ with $h \neq \ell$ if and only if there is at least one arc (i, j) in G with $i \in C_h$ and

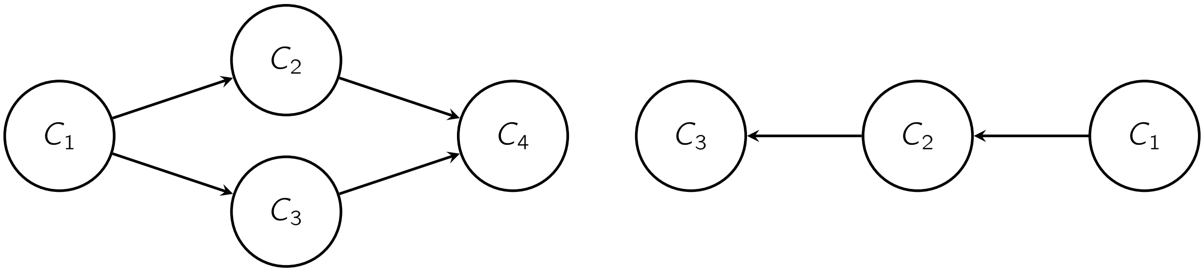


Figure 4: The DAGs of the SCCs of the graphs in Figs. 1 and 3.

$j \in C_\ell$. This directed graph must be acyclic: since within a SCC you can get from anywhere to anywhere else on a directed path, in a purported directed cycle of SCCs you can get from every node in a constituent SCC to every other node of every other SCC in the cycle. Thus the purported cycle of SCCs is actually just a single SCC. Summarizing, every directed graph has a useful "two-tier" structure: zooming out, one sees a DAG (Directed Acyclic Graph) on the SCCs of the graph; zooming in on a particular SCC exposes its finer-grained structure. For example, the meta-graphs corresponding to the directed graphs in Figs. 1 and 3 are shown in Fig. 4.

5 Proof of Correctness

5.1 The Key Lemma

Correctness of the algorithm hinges on the following key lemma.

Lemma 5. *Consider two "adjacent" strongly connected components of a graph G: components C_1 and C_2 such that there is an arc (i, j) of G with $i \in C_1$ and $j \in C_2$. Let $f(v)$ denote the finishing time of vertex v in some execution of DFS-Loop on the reversed graph G^{rev}. Then*

$$\max_{v \in C_1} f(v) < \max_{v \in C_2} f(v).$$

Proof. Consider two adjacent SCCs C_1 and C_2, as they appear in the reversed graph G^{rev} — where there is an arc (j, i), with $j \in C_2$ and $i \in C_1$ (Fig. 5). Because the equivalence relation defining the SCCs is symmetric, G and G^{rev} have the same SCCs; thus C_1 and C_2 are also SCCs of G^{rev}. Let v denote the first vertex of $C_1 \cup C_2$ visited by DFS-Loop in G^{rev}. There are now two cases.

First, suppose that $v \in C_1$ (Fig. 5). Since there is no non-trivial cycle of SCCs (Section 4.1), there is no directed path from v to C_2 in G^{rev}. Since DFS discovers everything reachable and nothing more, it will finish exploring all vertices in C_1 without reaching any vertices in C_2. Thus, *every* finishing time in C_1 will be smaller that *every* finishing time in C_2, and this is even stronger than the assertion of the lemma. (Cf., the left and middle SCCs in Fig. 3.)

Second, suppose that $v \in C_2$ (Fig. 5). Since DFS discovers everything reachable and nothing

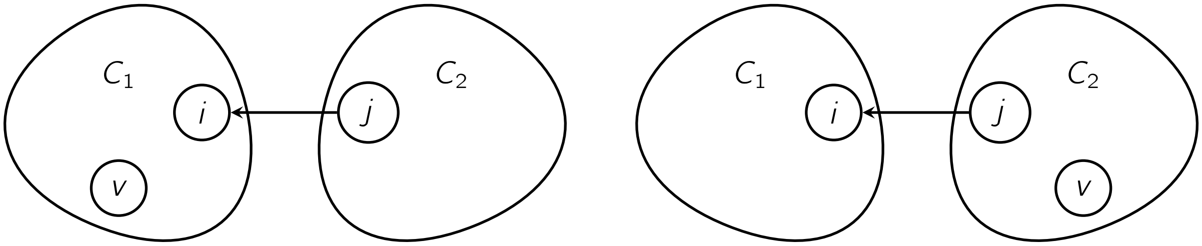


Figure 5: Proof of key lemma. Vertex v is the first in $C_1 \cup C_2$ visited during the execution of DFS-Loop on G^{rev}. On the left, all f-values in C_1 smaller than in C_2. On the right: v has the largest f-value in $C_1 \cup C_2$.

more, the call to DFS at v will finish exploring all of the vertices in $C_1 \cup C_2$ before ending. Thus, the finishing time of v is the largest amongst vertices in $C_1 \cup C_2$, and in particular is larger than all finishing times in C_1. (Cf., the middle and right SCCs in Fig. 3.)

This completes the proof.

□

5.2 The Final Argument

The Key Lemma says that traversing an arc from one SCC to another (in the original, unreversed graph) strictly increases the maximum f-value of the current SCC. For example, if f_i denotes the largest f-value of a vertex in C_i in Fig. 4, then we must have $f_1 < f_2, f_3 < f_4$. Intuitively, when DFS-Loop is invoked on G, processing vertices in decreasing order of finishing times, the successive calls to DFS peel off the SCCs of the graph one at a time, like layers of an onion.

We now formally prove correctness of our algorithm for computing strongly connected components. Consider the execution of DFS-Loop on G. We claim that whenever DFS is called on a vertex v, the vertices explored — and assigned a common leader — by this call are precisely those in v's SCC in G. Since DFS-Loop eventually explores every vertex, this claim implies that the SCCs of G are precisely the groups of vertices that are assigned a common leader.

We proceed by induction. Let S denote the vertices already explored by previous calls to DFS (initially empty). Inductively, the set S is the union of zero or more SCCs of G. Suppose DFS is called on a vertex v and let C denote v's SCC in G. Since the SCCs of a graph are disjoint, S is the union of SCCs of G, and $v \notin S$, no vertices of C lie in S. Thus, this call to DFS will explore, at the least, all vertices of C. By the Key Lemma, every outgoing arc (i, j) from C leads to some SCC C' that contains a vertex w with a finishing time larger than $f(v)$. Since vertices are processed in decreasing order of finishing time, w has already been explored and belongs to S; since S is the union of SCCs, it must contain all of C'. Summarizing, every outgoing arc from C leads directly to a vertex that has already been explored. Thus this call to DFS explores the vertices of C and nothing else. This completes the inductive step and the proof of correctness.

CS 161 (Stanford, Winter 2022) Lecture 11

Adapted from Virginia Williams's lecture notes. Additional credits go to Himanshu Bhandoh, Anthony Kim, Greg Valiant, Mary Wootters, and Aviad Rubinstein.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

Dijkstra and Bellman-Ford

1 Dijkstra's Algorithm

Now we will solve the single source shortest paths problem in graphs with nonnengative weights using Dijkstra's algorithm. The key idea, that Dijkstra will maintain as an invariant, is that $\forall t \in V$, the algorithm computes an estimate $d[t]$ of the distance of t from the source such that:

1. At any point in time, $d[t] \geq d(s,t)$, and
2. when t is finished, $d[t] = d(s,t)$.

Algorithm 1: Dijkstra($G = (V, E), s$)

```
$\forall t \in V, d[t] \leftarrow \infty$ // set initial distance estimates
$d[s] \leftarrow 0$
$F \leftarrow \{v \mid \forall v \in V\}$ // $F$ is set of nodes that are yet to achieve final distance estimates
$D \leftarrow \emptyset$ // $D$ will be set of nodes that have achieved final distance estimates
while $F \neq \emptyset$ do
    $x \leftarrow$ element in $F$ with minimum distance estimate
    for $(x, y) \in E$ do
        $d[y] \leftarrow \min\{d[y], d[x] + w(x,y)\}$ // "relax" the estimate of $y$
        // to maintain paths: if $d[y]$ changes, then $\pi(y) \leftarrow x$
    $F \leftarrow F \setminus \{x\}$
    $D \leftarrow D \cup \{x\}$
```

We will prove that Dijkstra correctly computes the distances from s to all $t \in V$.

Claim 1. *For every u, at any point of time $d[u] \geq d(s,u)$.*

A formal proof of this claim proceeds by induction. In particular, one shows that at any point in time, if $d[u] < \infty$, then $d[u]$ is the weight of some path from s to t. Thus at any point $d[u]$ is at least the weight of the *shortest* path, and hence $d[u] \geq d(s,u)$.

As a base case, we know that $d[s] = 0 = d(s,s)$ and all other distance estimates are $+\infty$, so we know that the claim holds initially. Now, when $d[u]$ is changed to $d[x] + w(x,u)$ then (by the induction hypothesis) there is a path from s to x of weight $d[x]$ and an edge (x,u) of weight $w(x,u)$. This means there is a path from s to u of weight $d[u] = d[x] + w(x,u)$. This implies that $d[u]$ is at least the weight of the shortest path $= d(s,u)$, and the induction argument is complete.

Claim 2. *When node x is placed in D, $d[x] = d(s, x)$.*

Notice that proving the above claim is sufficient to prove the correctness of the algorithm since $d[x]$ is never changed again after x is added to D: the only way it could be changed is if for some node $y \in F$, $d[y] + w(y, x) < d[x]$ but this can't happen since $d[x] \leq d[y]$ and $w(y, x) \geq 0$ (all edge weights are nonnegative). The assertion $d[x] \leq d[y]$ for all $y \in F$ stays true at all points after x is inserted into D: assume for contradiction that at some point for some $y \in F$ we get $d[y] < d[x]$ and let y be the first such y. Before $d[y]$ was updated $d[y'] \geq d[x]$ for all $y' \in F$. But then when $d[y]$ was changed, it was due to some neighbor y' of y in F, but $d[y'] \geq d[x]$ and all weights are nonnegative, so we get a contradiction

We prove this claim by induction on the order of placement of nodes into D. For the base case, s is placed into D where $d[s] = d(s, s) = 0$, so initially, the claim holds.

For the inductive step, we assume that for all nodes y currently in D, $d[y] = d(s, y)$. Let x be the node that currently has the minimum distance estimate in F (this is the node about to be moved from F to D). We will show that $d[x] = d(s, x)$ and this will complete the induction.

Let p be a shortest path from s to x. Suppose z is the node on p closest to x for which $d[z] = d(s, z)$. We know z exists since there is at least one such node, namely s, where $d[s] = d(s, s)$. By the choice of z, for every node y on p between z (not inclusive) to x (inclusive), $d[y] > d(s, y)$. Consider the following options for z.

1. If $z = x$, then $d[x] = d(s, x)$ and we are done.

2. Suppose $z \neq x$. Then there is a node z' after z on p. (Here it is possible that $z' = x$.) We know that $d[z] = d(s, z) \leq d(s, x) \leq d[x]$. The first $\leq$ inequality holds because subpaths of shortest paths are shortest paths as well, so that the prefix of p from s to z has weight $d(s, z)$. In addition, the weights on edges are non-negative, so that the portion of p from z to x has a nonnegative weight, and so $d(s, z) \leq d(s, x)$. The subsequent $\leq$ holds by Claim 1. We know that if $d[z] = d[x]$ all of the previous inequalities are equalities and $d[x] = d(s, x)$ and the claim holds.

   Finally, towards a contradiction, suppose $d[z] < d[x]$. By the choice of $x \in F$ we know $d[x]$ is the minimum distance estimate that was in F. Thus, since $d[z] < d[x]$, we know $z \notin F$ and must be in D, the finished set. This means the edges out of z, and in particular (z, z'), were already relaxed by our algorithm. But this means that $d[z'] \leq d(s, z) + w(z, z') = d(s, z')$, because z is on the shortest path from s to z', and the distance estimate of z' must be correct. However, this contradicts z being the closest node on p to x meeting the criteria $d[z] = d(s, z)$. Thus, our initial assumption that $d[z] < d[x]$ must be false and $d[x]$ must equal $d(s, x)$.

1.1 Implementation of Dijkstra's Algorithm

Consider implementing Dijkstra's algorithm with a priority queue to store the set F, where the distance estimates are the keys. The initialization step takes $O(n)$ operations to set n distance estimate values to infinity and 0. In each iteration of the while loop, we make a call to find the node x in F with the minimum distance estimate (via, say, `FindMin` operation). Then, we relax each edge leaving x (via `DecreaseKey`). We remove node x (via `DeleteMin`) and add it to D. In total, there are n calls to `FindMin` and n calls to `DeleteMin` since nodes are never re-inserted into F. Similarly, there will be m calls to `DecreaseKey` to relax the edges since each edge will be relaxed at most once.

Depending on how quickly our priority queue can support `FindMin`, `DeleteMin`, and `DecreaseKey` operations, the total runtime of Dijkstra's algorithm is on the order of

$$n \cdot (T_{\texttt{FindMin}}(n) + T_{\texttt{DeleteMin}}(n)) + m \cdot T_{\texttt{DecreaseKey}}(n).$$

We consider the following implementations of the priority queue for storing F:

- Store F as an array:
  Each slot corresponds to a node and stores the distance $d[j]$ if $j \in F$, or NIL otherwise. `DecreaseKey` runs in $O(1)$ as nodes are indexed. `FindMin` and `DeleteMin` run in $O(n)$ as the array is not sorted and we have to go through the whole array. The total runtime is $O(m + n^2) = O(n^2)$.

- Store F as a red-black tree:
  All operations run in $O(\log n)$ time. We implement `DecreaseKey` by deleting and re-inserting with the new key. The total runtime is $O((m + n)\log n)$. If graph G is sparse with few edges, then the red-black tree implementation is faster than the array implementation. However, it can be slower when G is dense with $m = \Theta(n^2)$.

- Store F as a Fibonacci heap:
  Fibonacci heaps are a complex data structure which is able to support the operations `Insert` in $O(1)$, `FindMin` in $O(1)$, `DecreaseKey` in $O(1)$ and `DeleteMin` in $O(\log n)$ "amortized" time, over a sequence of calls to these operations. The meaning of amortized time in this case is as follows: starting from an empty Fibonacci heap, any sequence of operations that includes a `Insert`'s, b `FindMin`'s, c `DecreaseKey`'s and d `DeleteMin`'s take $O(a + b + c + d \log n)$ time. The total runtime is $O(m + n \log n)$.

To conclude, Dijkstra's algorithm can be very fast when implemented the right way! However, it has a few drawbacks:

- It doesn't work with negative edge weights: we used the fact that the weights were non-negative a few times in the correctness proof above.

- It's not very amenable to frequent updates. Suppose that you had already run Dijkstra's algorithm from a particular point, but one weight in the graph changed. How would you recover from this? Next time, we'll see the Bellman-Ford algorithm, which can be better on both of these fronts.

2 Negative Edge Weights

Note that Dijkstra's algorithm solves the single source shortest paths problem when there are no edges with negative weights. While Dijkstra's algorithm may fail on certain graphs with negative edge weights, having a negative cycle (i.e., a cycle in the graph for which the sum of edge weights is negative) is a bigger problem for any shortest path algorithm. When computing a shortest path between two vertices, each additional traversal along the cycle lowers the overall cost incurred and an arbitrarily small distance can be reached after looping around the cycle multiple times. In this case, the shortest path to a node on the cycle is not well defined since it is (negatively) infinite.

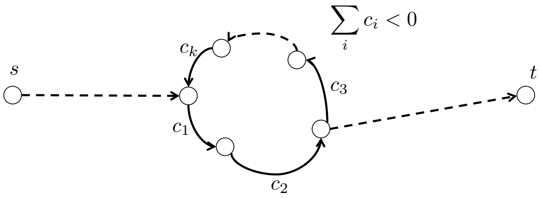


Figure 1: Assume there is a negative cycle along the $s - t$ path. The distance between s and t is not well-defined.

For example, consider the graph in Figure 1. The shortest path from s to t would start from the node s, loop around the negative cycle an infinite number of times and eventually reach destination t. The shortest path would, hence, be of infinite length and is not well-defined.

Besides the negative cycles, there are no problems in computing the shortest paths in a graph with negative edge weights. In fact, there are many applications where allowing negative edge weights is important.

3 Bellman-Ford Algorithm

In this section, we study the Bellman-Ford algorithm that solves the single source shortest paths problem on graphs with edges with potentially negative weights. Given a directed graph $G = (V, E)$ with edge weights given by $w(x, y)$ for $(x, y) \in E$, we want to compute the shortest path distances $d(s, v)$ from source s for all $v \in V$. More specifically, the Bellman-Ford algorithm:

- Detects a negative cycle if it exists and is reachable from s, or
- Computes the shortest path distances $d(s, v)$ for all $v \in V$.

Note $\pi(\cdot)$ is used to store the shortest paths found and $\pi(v)$ represents the predecessor of v on the shortest path from s to v.

NOTE: This version of Bellman-Ford is a bit different than the one we presented in class! As mentioned in class, we changed it up slightly to be more in line with the next lecture on

```
Algorithm 2: Bellman-Ford Algorithm
$d[v] \leftarrow \infty, \forall v \in V$   // set initial distance estimates
// to maintain paths: set $\pi(v) \leftarrow$nil for all $v$, $\pi(v)$ represents the predecessor of $v$
$d[s] \leftarrow 0$ // set distance to start node trivially as 0
for $i$ from $1 \rightarrow n-1$ do
    for $(u, v) \in E$ do
        $d[v] \leftarrow \min\{d[v], d[u] + w(u, v)\}$   // update the distance estimate for $v$
        // to maintain paths - if $d[v]$ changes, then $\pi(v) \leftarrow u$
// Negative Cycle Step
for $(u, v) \in E$ do
    if $d[v] > d[u] + w(u, v)$ then
        return "Negative Cycle"; // negative cycle detected
return $d[v]$ $\forall$ $v \in V$
```

Dynamic Programming. However, the analysis is basically the same. We'll analyze the above version here.

For an example run of the Bellman-Ford algorithm, please refer to the lecture slides or CLRS.

The total runtime of the Bellman-Ford algorithm is $O(mn)$. In the first for loop, we repeatedly update the distance estimates $n-1$ times on all m edges in time $O(mn)$. In the second for loop, we go through all m edges to check for negative cycles in time of $O(m)$.

We prove the correctness of the Bellman-Ford algorithm in two steps:

Claim 3. *If there is a negative cycle reachable from s, then the Bellman-Ford algorithm detects and reports "Negative Cycles".*

Proof. For the sake of contradiction, suppose there exists a negative cycle C reachable from the source s and the Bellman-Ford algorithm does not report "Negative Cycles". Assume C contains nodes $v_1, v_2, \ldots, v_k$ with edges (v_i, v_{i+1}) for $i = 1, \ldots, k$ such that $\sum_{i=1}^{k} w(v_i, v_{i+1}) < 0$, where $v_{k+1} = v_1$. See Figure 2. Let $d[\cdot]$ be the distance estimates determined in the first for loop of the algorithm.

Since C is reachable from s, there is a path from s to v_1 and to all nodes on C. In particular, there exist simple paths, i.e., paths without cycles, of at most $n-1$ edges to the nodes of C. In the first for loop, the edges on each such simply path get relaxed in order and consequently, $d[v_i]$ will be some finite number less than ∞ for $i = 1, \ldots, k$. Since the Bellman-Ford algorithm does not report "Negative Cycles" in the second for loop, it must be that $d[v_{i+1}] \leq d[v_i] + w(v_i, v_{i+1})$ for $i = 1, \ldots, k$. Adding the inequalities, we obtain

$$\sum_{i=1}^{k} d[v_{i+1}] \leq \sum_{i=1}^{k} d[v_i] + \sum_{i=1}^{k} w(v_i, v_{i+1}) \quad .$$

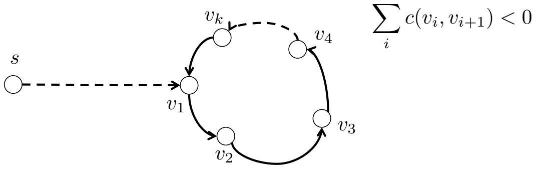


Figure 2: A negative cycle reachable from source s

As we are summing over the cycle C, the terms $\sum_{i=1}^{k} d[v_{i+1}]$ and $\sum_{i=1}^{k} d[v_i]$ are equal and can be cancelled. It follows that $0 \leq \sum_{i=1}^{k} w(v_i, v_{i+1})$. This contradicts that C is a negative cycle.

□

In the next claim, we show that if the graph has no negative cycles reachable from the source, then the Bellman-Ford algorithm returns the correct shortest path distances.

Claim 4. *If G has no negative cycles reachable from s, then $d[v] = d(s, v), \forall v \in V$.*

Proof. Let $d_k(v)$ be the value of $d[v]$ after k iterations of the first for loop. We prove by induction the statement that $d_k(v)$ is at most the minimum distance of a path from s to v with at most k edges. Then, we will have $d_{n-1}(v) = d[v]$ for all node v at termination. We'll argue below that if there is any path from s to v, then there is some shortest path with at most $n-1$ edges, so this means

$$\text{dist}(s, v) \leq d[v] = d_{n-1}(v) \leq \text{minimum cost of a path with at most } n-1 \text{ edges} = \text{dist}(s, v).$$

Thus, everything in the above inequality chain is equal, and in particular $d[v]$ is equal to the distance from s to v. Above, we used the fact that $d[v]$ is an over-estimate on $\text{dist}(s, v)$, which follows from our analysis of Dijkstra's algorithm.

We now argue that if there is a path from s to v, then there exists a shortest path from s to v has at most $n-1$ edges. If a shortest path has a cycle, the cycle cannot be negative and we can remove it and improve its total distance. If the cycle has a positive weight, removing the cycle will strictly improve the shortest path's distance. If the cycle has zero weight, we can ignore the cycle. Hence, we can assume that shortest paths are simple, that is, do not have cycles.

Base Case: When $k = 0$, the distance estimates have been just initialized. So, $d_0(v) = \infty$ if $v \neq s$. Furthermore, $d_0(s) = 0 = d(s, s)$, which is the minimum distance of length-0 paths from s to s. The statement is satisfied for the base case.

Inductive Step: Assume that $d_{k-1}(v)$ is at most the minimum distance of a $s \to v$ path on at most $k-1$ edges for all v.

Consider $v \neq s$. Let P be a shortest simple $s \to v$ path on at most k edges. Let u be the node just before v on P, and let Q be the sub-path of P from s to u. The path Q would have at most $k-1$ edges and is a shortest path from s to u with at most $k-1$ edges, since sub-paths of shortest paths are also shortest paths. By the inductive hypothesis, Q has cost at most $d_{k-1}(u)$.

In the k-th iteration, we update $d_k(v)$ such that $d_k(v) \leq d_{k-1}(u) + w(u,v) \leq w(Q) + w(u,v) = w(P)$.

The induction is complete, and the claim is proved. □

4 Amortized Time

Let's return to the Fibonacci heaps that we only very briefly mentioned above.

Note the runtimes listed for the operations of Fibonacci heaps are not worst-case runtimes. Instead, they are, what we call *amortized* runtimes. We say an operation on a data structure takes amortized $t(n)$ time if starting from an empty data structure, performing the operation L times takes $O(L \times t(n))$ time in total. This means the runtime of the operation is $O(t(n))$ when averaged over the sequence of L instances of the operation. Each individual operation call may take much more than $t(n)$ time, but this is compensated by many cheap operation calls (that take much less than $t(n)$ time).

We analyze the amortized cost of incrementing a *binary counter* by one when the count is represented in binary. Consider a b-bit counter which starts at 0 (i.e. b 0's). In each increment operation, we update the counter's bits correspondingly by flipping some bits from 0 to 1, or vice versa.

Some of the increment operations may take $\Omega(b)$ time. For example, an increment operation can require carrying b bits:

$$\begin{aligned} &11111111 \\ &+1 \\ =\ &100000000 \end{aligned}$$

Other increment operations can take $O(1)$ time:

$$\begin{aligned} &10000000 \\ &+1 \\ =\ &10000001 \end{aligned}$$

All this said, we can show the amortized cost of the increment operation on a binary counter is $O(1)$. Even though some increments take time linear in the number of bits, if we do n increment operations to the counter starting from the all 0s, each operation takes $O(1)$ time on average.

Claim 5. *The total time to increment a binary counter n times is $O(n)$.*

We use what is known as the *accounting* method to prove this claim. Each nonzero bit in the binary counter will get a "credit" obtained from earlier increment operations that will then be used to pay for later expensive operations. More specifically, we will maintain the *invariant* that every 1 in the binary representation has a "credit", which we represent as $\oplus$, associated with it.

Let x be the binary counter. If we start with an "empty" integer – that is 0 – then clearly all 1's have a credit as there are no 1's. Assume that all the 1's of x have a credit at the start of an increment operation. In each increment operation, we know the first addition will require constant work for which the addition operation will be charged with. We actually "charge" the addition operation *two* credits, represented as $\oplus\oplus$, to the new 1 to be added:

$$\begin{aligned} x = 1^{\oplus}1^{\oplus}&0 \\ &+ \\ &1^{\oplus\oplus} \end{aligned}$$

Now we start adding. We will maintain the invariant that any "carry" bit will have two $\oplus$ credits. For completeness, we'll call the original 1 to be added to x a "carry" as well.

Now, at each point we are adding a carry bit to a bit in x. If the carry bit is 0, we do nothing and stop. If the carry bit to be added to the i-th bit is 1 and the i-th bit of x is 0 (Note i starts at 0), then one of the $\oplus$ credits of the carry bit is used to store 1 in $x[i]$ and the other remains on this new 1 as $\oplus$:

$$\begin{aligned} &1^{\oplus}1^{\oplus}0 \\ &\quad +1^{\oplus\oplus} \\ = &1^{\oplus}1^{\oplus}1^{\oplus} \end{aligned}$$

At this point, the carry for the $i+1$-st slot is 0 and we can stop the addition.

When the carry bit to be added to the i-th bit is 1 and $x[i]$ is 1, however, we will get a non-zero carry bit for the $i+1$-st position. In this case, we will use one $\oplus$ from the 1 stored in $x[i]$ to pay for storing a 0 in $x[i]$ (doing the carry addition), and we'll move the two $\oplus$s of the carry bit to the new carry bit for the $i+1$-st position. This maintains the invariant that all 1s in x have a credit and all carries have two credits.

For example, consider an increment operation on the binary counter $x = 0111$:

$$\begin{aligned} &01^{\oplus}1^{\oplus}1^{\oplus} \\ &\quad +1^{\oplus\oplus} \end{aligned}$$

A new carry bit is formed:

$$\begin{aligned} &\quad 1^{\oplus\oplus} \\ &= 01^{\oplus}1^{\oplus}0 \end{aligned}$$

A new carry bit is formed:

$$\begin{aligned} &\quad 1^{\oplus\oplus} \\ &= 01^{\oplus}00 \end{aligned}$$

A new carry bit is formed:

$$\begin{aligned} &\quad 1^{\oplus\oplus} \\ &=0000 \\ &= 1^{\oplus}000 \ , \end{aligned}$$

arriving at $x = 1000$.

All carry propagations of additions are for free because they are paid for by the credits accumulated in previous additions of 0's and 1's. There are $O(n)$ credits overall, two for each increment operation. Thus, the total runtime is $O(n)$. The credit system allows you to pay for later long operations by depositing credits from previous short operations. Some operations are long, but over all n increment operations, the total work is $O(n)$. It follows that the increment operation takes amortized $O(1)$ time.

CS 161 (Stanford, Winter 2022) Lecture 12

Adapted from Virginia Williams's lecture notes. Additional credits go to Eric Huang, Anthony Kim, Mary Wootters, and Aviad Rubinstein.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

Dynamic Programming: Bellman-Ford and Floyd-Warshall

1 More on the Bellman-Ford Algorithm

We didn't quite make it to the Bellman-Ford algorithm in the last lecture, so we'll re-hash some of that again today. In the notes for the previous lecture, we introduced Bellman-Ford in the context of Dijkstra's algorithm. We'll see it in this lecture in a different way, so as to naturally introduce *dynamic programming.* The Bellman-Ford algorithm is a dynamic programming algorithm, and dynamic programming is a basic paradigm in algorithm design used to solve problems by relying on intermediate solutions to smaller subproblems. The main step for solving a dynamic programming problem is to analyze the problem's **optimal substructure** and **overlapping subproblems**.

The Bellman-Ford algorithm is pretty simple to state:

Algorithm 1: Bellman-Ford Algorithm (G, s)

```
$d^{(0)}[v] = \infty \forall v \in V$
$d^{(0)}[s] = 0$
$d^{(k)}[v] = \text{None} \forall v \in V \forall k > 0$
for $k = 1, \dots, n-1$ do
    $d^{(k)}[v] \leftarrow d^{(k-1)}[v]$ for all $v$
    for $(u, v) \in E$ do
        $d^{(k)}[v] \leftarrow \min\{d^{(k)}[v], d^{(k-1)}[u] + w(u, v)\}$
    Here we can release the memory for $d^{(k-1)}$, we'll never need it again.
return $d^{(n-1)}[v], \forall\ v \in V$
```

What's going on here? The value $d^{(k)}[v]$ is the cost of the shortest path from s to v with at most k edges in it. Once we realize this, a proof by induction (similar to the one in Lecture Notes 11) falls right out, with the inductive hypothesis that "$d^{(k)}[v]$ is the cost of the shortest path from s to v with at most k edges in it."

Runtime and Storage. The runtime of the Bellman-Ford algorithm is $O(mn)$; for n iterations, we loop through all the edges. This is slower than Dijkstra's algorithm. However, it is simpler to implement, and further as we saw in Lecture Notes 11.5, it can handle negative edge weights. For storage, in the pseudocode above, we keep n different arrays $d^{(k)}$ of length n. This isn't necessary: we only need to store two of them at a time. This is noted in the comment in the pseudocode.

1.1 What's really going on here?

The thing that makes that Bellman-Ford algorithm work is that that the shortest paths of length at most k can be computed by leveraging the shortest paths of length at most $k-1$. More specifically, we relied on the following recurrence relation between the intermediate solutions:

$$d^{(k)}[v] = \min_{u \in V} \left\{ d^{(k-1)}[u] + w(u, v) \right\}$$

where $d_k[v]$ is the length of the shortest path from source s to node v using at most k edges, and $w(u, v)$ is the weight of edge (u, v). (Above, we are assuming $w(v, v) = 0$).

This idea of using the intermediate solutions is similar to the divide-and-conquer paradigm. However, a divide-and-conquer algorithm recursively computes intermediate solutions once for each subproblem, but a dynamic programming algorithm solves the subproblems exactly once and uses these results multiple times.

2 Dynamic Programming

The idea of dynamic programming is to have a table of solutions of subproblems and fill it out in a particular order (e.g. left to right and top to bottom) so that the contents of any particular table cell only depends on the contents of cells before it. For example, in the Bellman-Ford algorithm, we filled out $d^{(k-1)}$ before we filled out $d^{(k)}$; and in order to fill out $d^{(k)}$, we just had to look back at $d^{(k-1)}$, rather than compute anything new.

In this lecture, we will discuss dynamic programming more, and also see another example: the Floyd-Warshall algorithm.

2.1 Dynamic Programming Algorithm Recipe

Here, we give a general recipe for solving problems (usually optimization problems) by dynamic programming. Dynamic programming is a good candidate paradigm to use for problems with the following properties:

- Optimal substructure gives a recursive formulation; and
- Overlapping subproblems give a small table, that is, we can store the precomputed answers such that it doesn't actually take too long when evaluating a recursive function multiple times.

What exactly do these things mean? We'll discuss them a bit more below, with the Bellman-Ford algorithm in mind as a reference.

2.1.1 Optimal Substructure

By this property, we mean that the optimal solution to the problem is composed of optimal solutions to smaller *independent* subproblems.

For example, the shortest path from s to t consists of a shortest path P from s to k (for node k on P) and a shortest path from k to t. This allows us to write down an expression for the distance between s and t with respect to the lengths of sub-paths:

$$d(s,t) = d(s,k) + d(k,t), \text{ for all } k \text{ on a shortest } s - t \text{ path}$$

We used this in the Bellman-Ford algorithm when we wrote

$$d^{(k)}[u] = \min_{v \in V} \{d^{(k-1)}[v] + w(u,v)\}.$$

2.1.2 Overlapping subproblems

The goal of dynamic programming is to construct a table of entries, where early entries in the table can be used to compute later entries. Ideally, the optimal solutions of subproblems can be reused multiple times to compute the optimal solutions of larger problems.

For our shortest paths example, $d(s,k)$ can used to compute $d(s,t)$ for any t where the shortest $s - t$ path contains k. To save time, we can compute $d(s,k)$ once and just look it up each time, instead of recomputing it.

More concretely in the Bellman-Ford example, suppose that (v,u) and (v,u') are both in E. When we go to compute $d^{(k)}[u]$, we'll need $d^{(k-1)}[v]$. Then when we go to compute $d^{(k)}[u']$, we'll need $d^{k-1}[v]$ again. If we just set this up as a divide-and-conquer algorithm, this would be extremely wasteful, and we'd be re-doing lots of work. By storing this value in a table and looking it up when we need it, we are taking advantage of the fact that these subproblems overlap.

2.1.3 Implementations

The above two properties lead to two different ways to implement dynamic programming algorithms. In each, we will store a table T with optimal solutions to subproblems; the two variants differ in how we decide to fill up the table:

1. Bottom-up: Here, we will fill in the table starting with the smallest subproblems. Then, assuming that we have computed the optimal solution to small subproblems, we can compute the answers for larger subproblems using our recursive optimal substructure.

2. Top-down: In this approach, we will compute the optimal solution to the entire problem recursively. At each recursive call, we will end up looking up the answer or filling in the table if the entry has not been computed yet.

In fact, these two methods are completely equivalent. Any dynamic programming algorithm can be formulated as an iterative table-filling algorithm or a recursive algorithm with look-ups.

3 Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm solves the All Pairs Shortest Path (APSP) problem: given a graph G, find the shortest path distances $d(s,t)$ for all $s,t \in V$, and, for the purpose of storing the shortest paths, the predecessor $\pi(s,t)$ which is the node right before t on the s-t shortest path.

Let's speculate about APSP for a moment. Consider the case when the edge weights are nonnegative. We know we can compute APSP by running Dijkstra's algorithm on each node $v \in V$ and obtain a total runtime of $O(mn + n^2 \log n)$. The runtime of the Floyd-Warshall algorithm, on the other hand, is $O(n^3)$. We know that in the worst case $m = O(n^2)$, and thus, the Floyd-Warshall algorithm can be at least as bad as running Dijkstra's algorithm n times! Then why do we care to explore this algorithm? The reason is that the Floyd-Warshall algorithm is very easy to implement compared to Dijkstra's algorithm. The benefit of using simple algorithms is that they can often be extended and in practice can run relatively faster compared to algorithms that may have a huge overhead.

An added benefit of the Floyd-Warshall algorithm is that it also supports negative edge weights, whereas Dijkstra's algorithm does not. [1]

As mentioned, the optimum substructure with overlapping subproblems for shortest paths is that for all node k on an s-t shortest path, $d(s,t) = d(s,k) + d(k,t)$. We refine this observation as follows. Suppose that the nodes of the graph are identified with the integers from 1 to n. Then, if k is the maximum node on an s-t shortest path, then $d(s,t) = d(s,k) + d(k,t)$ and moreover, the subpaths from s to k and from k to t only use nodes up to $k-1$ internally.

We hence get independent subproblems in which we compute $d_k(s,t)$ for all s,t that are the smallest weight of an s-t path that only uses nodes $1,\ldots,k$ internally. This motivates the Floyd-Warshall algorithm, Algorithm 2 below (please note that we will refer to the nodes of G by the names $1,\ldots,n$).

Correctness when there are no negative cycles In the k-th iteration of the Floyd-Warshall algorithm, $d_k(u,v)$ is the minimum weight of a $u \to v$ path that uses as intermediate nodes only nodes from $\{1,\ldots,k\}$. What does the recurrence relation represent? If P is a shortest path from u to v using $1,\ldots,k$ as intermediate nodes, there are two cases. Assume that P is a simple path, since shortest paths are simple when there are no negative cycles:

- *Case 1*: P contains k : In this case, we know that neither the path from u to k nor the path from k to v contains any nodes that are greater than $k-1$. In this case, we can simply use $d_k(u,v) = d_{k-1}(u,k) + d_{k-1}(k,v)$.
- *Case 2*: P does not contain k : We can say that $d_k(u,v) = d_{k-1}(u,v)$

[1] Although, one can still use Dijkstra's algorithm n times, if one preprocesses the edge weights initially via something called the Johnson's trick.

Algorithm 2: Floyd-Warshall Algorithm (G)

```
$d_k(u,u) = 0, \forall u \in V, k \in \{0,\dots,n\}$
$d_k(u,v) = \infty, \forall u,v \in V, u \neq v, k \in \{1,\dots,n\}$
$d_0(u,v) = c(u,v), \forall (u,v) \in E$
$d_0(u,v) = \infty, \forall (u,v) \notin E$
for $k = 1,\dots,n$ do
    for $(u,v) \in V$ do
        $d_k(u,v) = \min\{d_{k-1}(u,v), d_{k-1}(u,k) + d_{k-1}(k,v)\}$ // update the estimate of $d(u,v)$
return $d_n(u,v), \forall\ u,v \in V$
```

We initialize each $d_0(u,v)$ as the edge weight $c(u,v)$ if $(u,v) \in E$, else we set it to ∞ in the bottom-most row in our dynamic programming table. Now, as we increment k to 1, we effectively find the minimum distance path between $u,v \in V$ that go through node 1, and populate the table with the results. We continue this process to find the shortest paths that go through nodes 1 and 2, then $1,2$, and 3 and so on until we find the shortest path through all n nodes.

Negative cycles. The Floyd-Warshall algorithm can be used to detect negative cycles: examine whether $d_n(u,u) < 0$ for any $u \in V$. If there exists u such that $d_n(u,u) < 0$, there is a negative cycle, and if not, then there isn't. The reason for this is that if there is a simple path P from u to u of negative weight (i.e., a negative cycle containing u), then $d_n(u,u)$ will be at most its weight, and hence, will be negative. Otherwise, no path can cause $d_n(u,u)$ to be negative.

Runtime. The runtime of the Floyd-Warshall algorithm is proportional to the size of the table $\{d_i(u,v)\}_{i,u,v}$ since filling each entry of the table only depends on at most two other entries filled in before it. Thus, the runtime is $O(n^3)$.

Space usage. Note that for both the algorithms we covered today, the Floyd-Warshall and Bellman-Ford algorithms, we can choose to store only two rows of the table instead of the complete table in order to save space. This is because the row being populated always depends only on the row right below it. This space saving optimization is not a general property of tables formed as a result of the dynamic programming method, and the slot dependencies in some dynamic programming problems may lie on arbitrary positions on the table thereby forcing us to store the complete table.

A Note on the Longest Path Problem

We discussed the shortest path problem in detail and provided algorithms for a number of variants of the problem. We might equally be interested in computing the longest simple path in a graph. A first approach is to formulate a dynamic programming algorithm. Indeed, consider any path, even the longest, between two nodes s and t. Its length $\ell(s,t)$ equals the sum $\ell(s,k) + \ell(k,t)$ for any node k on the path. However, this does not yield an optimal substructure: in general, neither subpath $s \to k$, $k \to t$ would be a longest path, and even if one is a longest path, the other one cannot use any nodes that appear on the first since the longest path is required to be simple. Hence the two subproblems $\ell(s,k)$ and $\ell(k,t)$ are not even independent! It turns out that finding the longest path does not seem to have any optimal substructure, which makes it difficult to avoid exhaustive search through dynamic programming. The longest path problem is actually a very difficult problem to solve and is NP-hard. The best known algorithm for it runs in exponential time.

4 Why is it called dynamic programming?

The name doesn't immediately make a lot of sense. “Dynamic programming" sounds like the type of coding that action heroes do in late-90's hacker movies. However, “progamming" here refers to a program, like a plan (for example, the path you are trying to optimize), not to programming a computer. “Dynamic" refers to the fact that we update the table over time: this is a dynamic process. But the fact that it makes you (or at least me) think about action movies isn't an accident. As Richard Bellman, who coined the term, writes in his autobiography:

> An interesting question is, "Where did the name, dynamic programming, come from?" The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. Im not using the term lightly; Im using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place, I was interested in planning, in decision-making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying- I thought, let's kill two birds with one stone. Let's take a word which has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in the pejorative sense. Try thinking of some combination which will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

CS 161 (Stanford, Winter 2022) Lecture 13

Adapted from Virginia Williams' lecture notes. Additional credits go to Wilbur Yang and Mary Wootters.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

More Dynamic Programming

1 Overview

Last lecture, we talked about dynamic programming (DP), a useful paradigm and one technique that you should immediately consider when you are designing an algorithm. We covered the Bellman-Ford algorithm for solving the single source shortest path problem, and we talked about the Floyd-Warshall algorithm for solving the all pairs shortest path problem.

This lecture, we will cover some more examples of dynamic programming, and start to see a recipe for how to come up with DP solutions. We will talk about three problems today: longest common subsequence, knapsack, and maximum weight independent set in trees.

In general, here are the steps to coming up with a dynamic programming algorithm:

1. **Identify optimal substructure:** how are we going to break up an optimal solution into optimal sub-solutions of sub-problems? We're looking for a way to do this so that there are *overlapping* sub-problems, so that a dynamic programming approach will be effective.

2. **Recursively define the value of an optimal solution:** Write down a recursive formulation of the optimum, in terms of sub-solutions.

3. **Find the optimal value:** Turn this recursive formulation into a dynamic programming algorithm to compute the value of the optimal solution.

4. **Find the optimal solution:** Once we've figured out how to find the cost of the optimal solution, we can go back and figure out how to keep enough information in our algorithm so that we can find the solution itself.

5. **Tweak the implementation:**[1] Often it's the case that the solutions that we come up with in the previous steps aren't implemented in the best way. Maybe they are storing more than they need to, like we saw with our first pass at the Floyd-Warshall algorithm. In this final step (which we won't go into in too much detail in CS161), we go back through the DP solution we've designed, and optimize it for space, running time, and so on.

[1] We won't talk too much about this step in CS161, even though it is often important in practice.

In this class, we'll focus mostly on 1, 2, and 3. We'll see a few examples of 4, and occasionally wave our hands about 5.

2 Longest Common Subsequence

We now consider the longest common subsequence problem which has applications in spell-checking, biology (whether different DNA sequences correspond to the same protein), and more.

We say that a sequence Z is a *subsequence* of a sequence X if Z can be obtained from X by deleting symbols. For example, `abracadabra` has `baab` as a subsequence, because we can obtain `baab` by deleting `a`, `r`, `cad`, and `ra`. We say that a sequence Z is a *longest common subsequence* (LCS) of X and Y if Z is a subsequence of both X and Y, and any sequence longer than Z is not a subsequence of at least one of X or Y. For instance, the LCS of `abracadabra` and `bxqrabry` is `brabr`.

Using the definition of LCS, we define the LCS problem as follows: Given sequences X and Y, find the length of their LCS, Z (and if we are proceeding to Step 4 of the outline above, output Z).

In what follows, suppose that the sequence X is $X = x_1 x_2 x_3 \cdots x_m$, so that X has length m, and suppose that $Y = y_1 y_2 \cdots y_n$ as length n. We'll use the notation $X[1:k]$ as usual to denote the *prefix* $X[1:k] = x_1 x_2 \cdots x_k$.

2.1 Steps 1 and 2: Identify optimal substructure, and write a recursive formulation

Our sub-problems will be to solve LCS on prefixes of X and Y. To see how we can do this, we consider the following two cases.

- **Case 1:** $x_m = y_n$**.** If $x_m = y_n = \ell$, then any LCS Z has ℓ as its last symbol. Indeed, suppose that Z' is any common subsequence that does *not* end in ℓ: then we can always extend it by appending ℓ to Z' to obtain another (longer) legal common subsequence.

  Thus, if $|Z| = k$ and $x_m = y_n = \ell$, we can write

  $$Z[1:k-1] = \mathrm{LCS}(X[1:m-1], Y[1:n-1])$$

  and

  $$Z = Z[1:k-1] \circ \ell,$$

  where $\circ$ denotes the concatenation operation on strings.

- **Case 2:** $x_m \neq y_n$**.** As above, let Z be the LCS of X and Y. In this case, the last letter of Z (call it z_k) is either not equal to x_m or it is not equal to y_n. (Notice that this or is not an exclusive or; maybe z_k isn't equal to either x_m or y_n). In this case, at least one of x_m or y_n cannot appear in the LCS of X and Y; this means that either

$$\mathrm{LCS}(X, Y) = \mathrm{LCS}(X[1:m-1], Y)$$

or

$$\mathrm{LCS}(X, Y) = \mathrm{LCS}(X, Y[1:n-1]),$$

whichever is longer. That is, we can shave one letter off the end of either X or Y. In particular, the length of LCS(X, Y) is given by

$$\mathrm{lenLCS}(X, Y) = \max\{\mathrm{lenLCS}(X[1:m-1], Y), \mathrm{lenLCS}(X, Y[1:n-1])\}.$$

This immediately gives us our recursive formulation. Let's keep a table C, so that

$$C[i,j] = \text{length of } \mathrm{LCS}(X[1:i], Y[1:j]).$$

Then we have the relationship:

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0, \\ \max\{C[i-1,j], C[i,j-1]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0. \end{cases}$$

Suppose we keep a table C, where $C[i,j]$ maintains the length of $LCS(X[1:i], Y[1:j])$, the longest common subsequence of $X[1:i]$ and $Y[1:j]$. Then, we can fill in the values of C using the following recurrence:

$$C[i,j] = \begin{cases} 1 + C[i-1,j-1], & \text{if } X[i] = Y[j], \\ \max(C[i-1,j], C[i,j-1]) & \text{otherwise.} \end{cases}$$

Technically, we should do a proof here to show that this recurrence is correct. See CLRS for the details, but it is true that if we define $C[i,j]$ recursively as above, then indeed, $C[i,j]$ is equal to the length of LCS($X[1:i], Y[1:j]$). (Good exercise: prove this for yourself using induction).

2.2 Step 3: Define an algorithm using our recursive relationship.

The recursive relationship above naturally gives rise to a DP algorithm for filling out the table C:

Note that there are only $n \times m$ entries in our table C. This is where the **overlapping subproblems** come in: we only need to compute each entry once, even though we may access it many times when filling out subsequent entries.

We can also see that $C[i,j]$ only depends on three possible prior values: $C[i-1,j]$, $C[i,j-1]$, and $C[i-1,j-1]$. This means that each time we compute a new value $C[i,j]$ from previous entries, it takes time $O(1)$.

Algorithm 1: lenLCS(X, Y)

```
Initialize an $n + 1 \times m + 1$ zero-indexed array $C$.
Set $C[0, j] = C[i, 0] = 0$ for all $i, j \in \{1, \ldots, m\} \times \{1, \ldots, n\}$.
for $i = 1, \ldots, m$ do
    for $j = 1, \ldots, n$ do
        if $X[i] = Y[j]$ then
            $C[i, j] \leftarrow C[i - 1, j - 1] + 1$
        else
            $C[i, j] \leftarrow \max\{C[i - 1, j], C[i, j - 1]\}$
return $C$
```

Thus, we can start to see how to obtain an algorithm for filling in the table and obtaining the LCS. First, we know that any string of length 0 will have an LCS of length 0. Thus, we can start by filling out $C[0, j] = 0$ for all j and similarly, $C[i, 0] = 0$ for all i. Then, we can fill out the rest of the table, filling the rows from bottom up (i from 1 to m) and filling each row from left to right (j from 1 to n). The pseudocode is given in Algorithm 1.

As mentioned above, in order to fill each entry, we only need to perform a constant number of lookups and additions. Thus, we need to do a constant amount of work for each of the $m \times n$ entries, giving a running time of $O(mn)$.

2.3 Step 4: Recovering the actual LCS

Algorithm 1 only computes the *length* of the LCS of X and Y. What if we want to recover the actual longest common subsequence? In Algorithm 2, we show how we can construct the actual LCS, given the dynamic programming table C that we've filled out in Algorithm 1.

In this algorithm, we start from the end of X and Y and work backward, using our table C as a guide. We start with $i = m$ and $j = m$. If at some point (i, j), we see that $X[i] = Y[j]$, then decrement both i and j. On the other hand, if $X[i] \neq Y[j]$, then we know that we need to drop a symbol from either X or Y. The table C will tell us which: if $C[i, j] = C[i, j - 1]$, then we can drop a symbol from Y and decrement j. If $C[i, j] = C[i - 1, j]$, then we can drop a symbol from X and decrement i. Of course, it might be the case that both of these hold; in this case it doesn't matter which we decrement, and our pseudocode will be default decrement j.

How long does this take? Notice that in each step, the sum $i + j$ is decremented by at least one (maybe two) and stops as soon as one of i, j is equal to zero; this is at least before $i + j = 0$. Thus, the number of times we decrement $i + j$ is at most $m + n$, which was their total value to start.

Because at each step of Algorithm 2, the work is $O(1)$, the total running time is thus $O(n+m)$, which is subsumed by the runtime of $O(mn)$ necessary to fill in the table.

Algorithm 2: LCS(X, Y, C)

```
// C is filled out already in Algorithm 1
$L \leftarrow \emptyset$
$i \leftarrow m$
$j \leftarrow n$
while $i > 0$ and $j > 0$ do
    if $X[i] = Y[j]$ then
        Append $X[i]$ to the beginning of $L$
        $i \leftarrow i - 1$
        $j \leftarrow j - 1$
    else if $C[i, j] = C[i, j - 1]$ then
        $j \leftarrow j - 1$
    else
        $i \leftarrow i - 1$
```

The conclusion is that we can find LCS(X, Y) of a sequence X of length m and a sequence Y of length n in time $O(mn)$.

Interestingly, this simple dynamic programming algorithm is basically the best known algorithm for solving the LCS problem. It is conjectured that this algorithm may be essentially optimal. It turns out that giving an algorithm that (polynomially) improves the dependence on m and n over the $O(mn)$ strategy outlined above would imply a major breakthrough in algorithms for the boolean satisfiability problem – a problem widely believed to be computationally hard to solve.

3 The Knapsack Problem

This is a classic problem, defined as the following:

We have n items, each with a value and a positive weight. The i-th item has weight w_i and value v_i. We have a knapsack that holds a maximum weight of W. Which items do we put in our knapsack to maximize the value of the items in our knapsack? For example, let's say that $W = 10$; that is, the knapsack holds a weight of at most 10. Also suppose that we have four items, with weight and value:

Item	Weight	Value
A	6	25
B	3	13
C	4	15
D	2	8

We will talk about two variations of this problem, one where you have infinite copies of each item (commonly known as Unbounded Knapsack), and one where you have only one of each item (commonly known as 0-1 Knapsack).

What are some useful subproblems? Perhaps it's having knapsacks of smaller capacities, or maybe it's having fewer items to choose from. In fact, both of these ideas for subproblems are useful. As we will see, the first idea is useful for the Unbounded Knapsack problem, and a combination of the two ideas is useful for the 0-1 Knapsack problem.

3.1 The Unbounded Knapsack Problem

In the example above, we can pick two of item B and two of item D. Then, the total weight is 10, and the total value 42.

We define $K(x)$ to be the optimal solution for a knapsack of capacity x. Suppose $K(x)$ happens to contain the i-th item. Then, the remaining items in the knapsack must have a total weight of at most $x - w_i$. The remaining items in the knapsack must be an optimum solution. (If not, then we could have replaced those items with a more highly valued set of items.) This gives us a nice subproblem structure, yielding the recurrence

$$K(x) = \max_{i : w_i \leq x} \left(K(x - w_i) + v_i \right) .$$

Developing a dynamic programming algorithm around this recurrence is straightforward. We first initialize $K(0) = 0$, and then we compute $K(x)$ values from $x = 1, \ldots, W$. The overall runtime is $O(nW)$.

Algorithm 3: UnboundedKnapsack(W, n, w, v)

```
$K[0] \leftarrow 0$
for $x = 1, \ldots, W$ do
    $K[x] \leftarrow 0$
    for $i = 1, \ldots, n$ do
        if $w_i \leq x$ then
            $K[x] \leftarrow \max\{K[x], K[x - w_i] + v_i\}$
return $K[W]$
```

Remark 1. This solution is not actually polynomial in the input size because it takes $\log(W)$ bits to represent W. We call these algorithms "pseudo-polynomial." If we had a polynomial time algorithm for Knapsack, then a lot of other famous problems would have polynomial time algorithms. This problem is NP-hard.

3.2 The 0-1 Knapsack Problem

Now we consider what happens when we can take at most one of each item. Going back to the initial example, we would pick item A and item C, having a total weight of 10 and a total value of 40.

The subproblems that we need must keep track of the knapsack size as well as which items are allowed to be used in the knapsack. Because we need to keep track of more information in our state, we add another parameter to the recurrence (and therefore, another dimension to the DP table). Let $K(x, j)$ be the maximum value that we can get with a knapsack of capacity x considering only items at indices from $1, \ldots, j$. Consider the optimal solution for $K(x, j)$. There are two cases:

1. Item j is used in $K(x, j)$. Then, the remaining items that we choose to put in the knapsack must be the optimum solution for $K(x - w_j, j - 1)$. In this case, $K(x, j) = K(x - w_j, j - 1) + v_j$.
2. Item j is not used in $K(x, j)$. Then, $K(x, j)$ is the optimum solution for $K(x, j - 1)$. In this case, $K(x, j) = K(x, j - 1)$.

So, our recurrence relation is: $K(x, j) = \max\{K(x - w_j, j - 1) + v_j, K(x, j - 1)\}$. Now, we're done: we simply calculate each entry up to $K(W, n)$, which gives us our final answer. Note that this also runs in $O(nW)$ time despite the additional dimension in the DP table. This is because at each entry of the DP table, we do $O(1)$ work.

Algorithm 4: ZeroOneKnapsack(W, n, w, v)

```
for $x = 1, \ldots, W$ do
    $K[x, 0] \leftarrow 0$
for $j = 1, \ldots, n$ do
    $K[0, j] \leftarrow 0$
for $j = 1, \ldots, n$ do
    for $x = 1, \ldots, W$ do
        $K[x, j] \leftarrow K[x, j - 1]$
        if $w_j \leq x$ then
            $K[x, j] \leftarrow \max\{K[x, j], K[x - w_j, j - 1] + v_j\}$
return $K[W, n]$
```

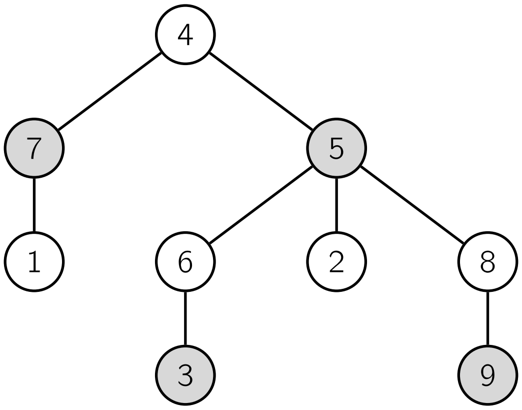
4 The Independent Set Problem

This problem is as follows:

Say that we have an undirected graph $G = (V, E)$. We call a subset $S \subseteq V$ of vertices "independent" if there are no edges between vertices in S. Let vertex i have weight w_i, and denote $w(S)$ as the sum of weights of vertices in S. Given G, find an independent set of maximum weight $\text{argmax}_{S \subseteq V} w(S)$.

Actually, this problem is NP-hard for a general graph G. However, if our graph is a tree, then we can solve this problem in linear time. In the following figure, the maximum weight independent set is highlighted in blue.

Remark 2. Dynamic programming is especially useful to keep in mind when you are solving a problem that involves trees. The tree structure often lends itself to dynamic programming solutions.



As usual, the key question to ask is, "What should our subproblem(s) be?" Intuitively, if the problem has to do with trees, then subtrees often play an important role in identifying our subproblems. Let's pick any vertex r and designate it as the root. Denoting the subtree rooted at u as T_u, we define $A(u)$ to be the weight of the maximum weight independent set in T_u. How can we express $A(u)$ recursively? Letting S_u be the maximum weight independent set of T_u, there are two cases:

1. If $u \notin S_u$, then $A(u) = \sum_v A(v)$ for all children v of u.
2. If $u \in S_u$, then $A(u) = w_u + \sum_v A(v)$ for all grandchildren v of u.

To avoid solving the subproblem for trees rooted at grandchildren, we introduce $B(u)$ as the weight of the maximum weight independent set in $T_u \setminus \{u\}$. That is, $B(u) = \sum_v A(v)$ for all children v of u. Equivalently, we have the following cases:

1. If $u \notin S_u$, then $A(u) = \sum_v A(v)$ for all children v of u.
2. If $u \in S_u$, then $A(u) = w_u + \sum_v B(v)$ for all children v of u.

So, we can calculate the weight of the maximum weight independent set:

$$A(u) = \max \left\{ \sum_{v \in \text{Children}(u)} A(v), w_u + \sum_{v \in \text{Children}(u)} B(v) \right\}$$

To create an algorithm out of this recurrence, we can compute the $A(u)$ and $B(u)$ values in a bottom-up manner (a post-order traversal on the tree), arriving at the answer, $A(r)$. This takes $O(|V|)$ time.

Algorithm 5: MaxWeightIndependentSet(G)

```
// G is a tree
$r \leftarrow$ ArbitraryVertex($G$)
$T \leftarrow$ RootTreeAt($G, r$)
Procedure SolveSubtreeAt($u$)
    if Children($T, u$) $= \emptyset$ then
        $A(u) \leftarrow w_u$
        $B(u) \leftarrow 0$
    else
        for $v \in$ Children($T, u$) do
            SolveSubtreeAt($v$)
        $A(u) \leftarrow \max \left\{ \sum_{v \in \mathrm{Children}(T,u)} A(v), w_u + \sum_{v \in \mathrm{Children}(T,u)} B(v) \right\}$
        $B(u) \leftarrow \sum_{v \in \mathrm{Children}(T,u)} A(v)$
SolveSubtreeAt($r$)
return $A(r)$
```

CS 161 (Stanford, Winter 2022) Lecture 14

Adapted from Virginia Williams' lecture notes. Additional credits go to Sam Kim and Mary Wootters.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

Greedy Algorithms

1 Greedy Algorithms

Suppose we want to solve a problem, and we're able to come up with some recursive formulation of the problem that would give us a nice dynamic programming algorithm. But then, upon further inspection, we notice that any optimal solution only depends on looking up the optimal solution to one other subproblem. A greedy algorithm is an algorithm which exploits such a structure, ignoring other possible choices. Greedy algorithms can be seen as a refinement of dynamic programming; in order to prove that a greedy algorithm is correct, we must prove that to compute an entry in our table, it is sufficient to consider at most one other table entry; that is, at each point in the algorithm, we can make a "greedy", locally-optimal choice, and guarantee that a globally-optimal solution still exists. Instead of considering multiple choices to solve a subproblem, greedy algorithms only consider a single subproblem, so they run extremely quickly – generally, linear or close-to-linear in the problem size.

Unfortunately, greedy algorithms do not always give the optimal solution, but they frequently give good (approximate) solutions. To give a correct greedy algorithm one must first identify optimal substructure (as in dynamic programming), and then argue that at each step, you only need to consider one subproblem. That is, even though there may be many possible subproblems to recurse on, given our selection of subproblem, there is always an optimal solution that contains the optimal solution to the selected subproblem.

1.1 Activity Selection Problem

One problem, which has a very nice (correct) greedy algorithm, is the Activity Selection Problem. In this problem, we have a number of activities. Your goal is to choose a subset of the activities to participate in. Each activity has a start time and end time, and you can't participate in multiple activities at once. Thus, given n activities $a_1, a_2, \ldots, a_n$ where a_i has start time s_i and finish time f_i, we want to find a maximum set of non-conflicting activities.

The activity selection problem has many applications, most notably in scheduling jobs to run on a single machine.

1.1.1 Optimal Substructure

Let's start by considering a subset of the activities. In particular, we'll be interested in considering the set of activities $S_{i,j}$ that start after activity a_i finishes and end before activity a_j starts. That is, $S_{i,j} = \{a_k \mid f_i \leq s_k, f_k \leq s_j\}$. We can participate in these activities between a_i and a_j. Let $A_{i,j}$ be a maximum subset of non-conflicting activities from the subset $S_{i,j}$. Our first intuition would be to approach this by using dynamic programming. Suppose some $a_k \in A_{i,j}$, then we can break down the optimal subsolution $A_{i,j}$ as follows

$$|A_{i,j}| = 1 + |A_{i,k}| + |A_{k,j}|$$

where $A_{i,k}$ is the best set for $S_{i,k}$ (before a_k), and $A_{k,j}$ is the best set for after a_k. Another way of interpreting this expression is to say "once we place a_k in our optimal set, we can only consider optimal solutions to subproblems that do not conflict with a_k."

Thus, we can immediately come up with a recurrence that allows us to come up with a dynamic programming algorithm to solve the problem.

$$|A_{i,j}| = \max_{a_k \in S_{i,j}} 1 + |A_{i,k}| + |A_{k,j}|.$$

This problem requires us to fill in a table of size n^2, so the dynamic programming algorithm will run in $\Omega(n^2)$ time. The actual runtime is $O(n^3)$ since filling in a single entry might take $O(n)$ time.

But we can do better! We will show that we only need to consider the a_k with the smallest finishing time, which immediately allows us to search for the optimal activity selection in linear time.

Proposition 1. *For each $S_{i,j}$, there is an optimal solution $A_{i,j}$ containing $a_k \in S_{i,j}$ of minimum finishing time f_k.*

Note that if the proposition is true, when f_k is minimum, then $A_{i,k}$ is empty, as no activities can finish before a_k; thus, our optimal solution only depends on one other subproblem $A_{k,j}$ (giving us a linear time algorithm).

Here, we prove the proposition.

Proof. Let a_k be the activity of minimum finishing time in $S_{i,j}$. Let $A_{i,j}$ be some maximum set of non-conflicting activities. Consider $A'_{i,j} = A_{i,j} \setminus \{a_l\} \cup \{a_k\}$ where a_l is the activity of minimum finishing time in $A_{i,j}$. It's clear that $|A'_{i,j}| = |A_{i,j}|$. We need to show that $A'_{i,j}$ does not have conflicting activities. We know $a_l \in A_{i,j} \subset S_{i,j}$. This implies $f_l \geq f_k$, since a_k has the minimum finishing time in $S_{i,j}$.

All $a_t \in A_{i,j} \setminus \{a_l\}$ don't conflict with a_l, which means that $s_t \geq f_l$, which means that $s_t \geq f_k$, so this means that no activity in $A_{i,j} \setminus \{a_l\}$ can conflict with a_k. Thus, $A'_{i,j}$ is an optimal solution. □

Due to the above proposition, the expression for $A_{i,j}$ from before simplifies to the following expression in terms of $a_k \subseteq S_{i,j}$, the activity with minimum finishing time f_k.

$$|A_{i,j}| = 1 + |A_{k,j}|$$
$$A_{i,j} = A_{k,j} \cup \{a_k\}$$

Algorithm Greedy-AS assumes that the activities are presorted in nondecreasing order of their finishing time, so that if $i < j$, $f_i \leq f_j$.

Algorithm 1: Greedy-AS(a)

```
$A \leftarrow \{a_1\}$ /* activity of min $f_i$                              */
$k \leftarrow 1$
for $m = 2 \to n$ do
    if $s_m \geq f_k$ then
        // $a_m$ starts after last acitivity in $A$
        $A \leftarrow A \cup \{a_m\}$
        $k \leftarrow m$
return $A$
```

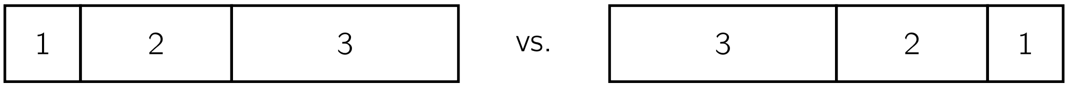
By the above claim, this algorithm will produce a legal, optimal solution via a greedy selection of activities. There may be multiple optimal solutions, but there always exists a solution that includes a_k with the minimum finishing time. The algorithm does a single pass over the activities, and thus only requires $O(n)$ time – a dramatic improvement from the trivial dynamic programming solution. If the algorithm also needed to sort the activities by f_i, then its runtime would be $O(n \log n)$ which is still better than the original dynamic programming solution.

1.2 Scheduling

Consider another problem that can be solved greedily. We are given n jobs which all need a common resource. Let w_j be the weight (or importance) and l_j be the length (time required) of job j. Our output is an ordering of jobs. We define the completion time c_j of job j to be the sum of the lengths of jobs in the ordering up to and including l_j. Our goal is to output an ordering of jobs that minimizes the weighted sum of completion times $\sum_j w_j c_j$.

1.2.1 Intuition

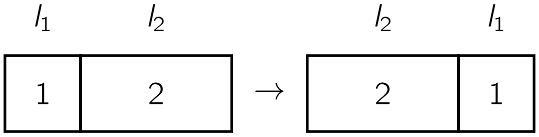
Our intuition tells us that if all jobs have the same length, then we prefer larger weighted jobs to appear earlier in the order. If jobs all have equal weights, then we prefer shorter length jobs in the order.



In the first case, assuming they all have equal weights of 1, $\sum_{i=1}^{3} w_i c_i = 1 + 3 + 6 = 10$. In the second case, $\sum_{i=1}^{3} w_i c_i = 3 + 5 + 6 = 14$.

1.2.2 Optimal Substructure

What do we do in the cases where $l_i < l_j$ and $w_i < w_j$? Consider the optimal ordering of jobs. Suppose we have a job i that is followed by job j in the optimal order. Consider swapping jobs i and j. The example below swaps jobs 1 and 2.



Note that swapping jobs i and j does not alter the completion times for every other job and only changes the completion times for i and j. c_i increases by l_j and c_j decreases by l_i. This means that our objective function $\sum_i w_i c_i$ changes by $w_i l_j - w_j l_i$. Since we assumed our order was optimal originally, our objective function cannot decrease after swapping the jobs. This means,

$$w_i l_j - w_j l_i \geq 0$$

which implies

$$\frac{l_j}{w_j} \geq \frac{l_i}{w_i}.$$

Therefore, we want to process jobs in increasing order of $\frac{l_i}{w_i}$, the ratio of the length to the weight of each job. The algorithm also does a single pass over jobs, and thus only requires $O(n)$ time, assuming the jobs were ordered by $\frac{l_i}{w_i}$. Like previously, if the algorithm also needed to sort the jobs based on the ratio of length to weight, then its runtime would be $O(n \log n)$.

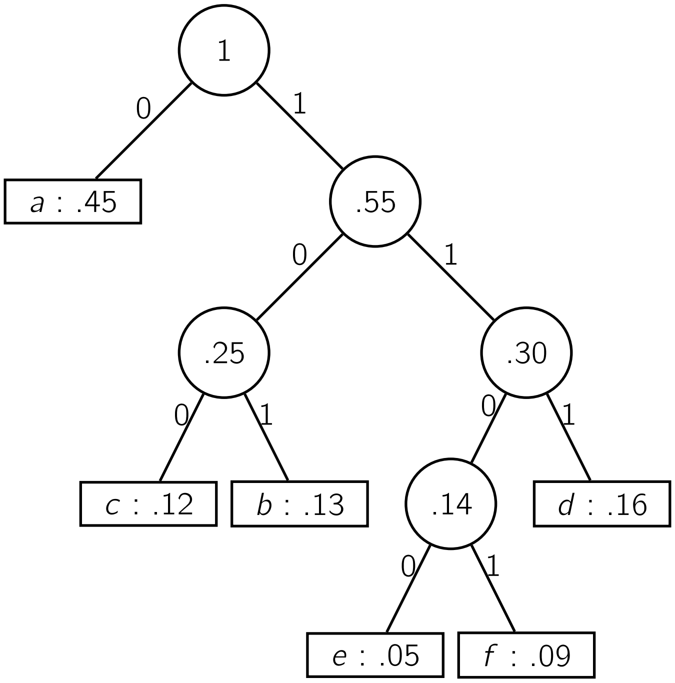
1.3 Optimal Codes

Our third example comes from the field of information theory. In ASCII, there is a fixed 8 bit code for each character. Suppose we want to incorporate information about frequencies of characters to obtain shorter encodings. What if we want to represent characters by codes of different lengths depending on each character's frequencies? We explore a greedy solution to find the optimal encoding of characters.

To create optimal codes, we want a way to encode and decode our sequence. To encode the sequence, we would just have to concatenate the code of each character together. How about for decoding? Consider the following codes of characters: $a \rightarrow 0, b \rightarrow 1, c \rightarrow 01$. However, when decoding, when we encounter 01, this could be decoded as "ab" or "c". Therefore, our codes need to be **prefix free**: no codeword is a prefix of another.

1.3.1 Tree Representation

We may think of representing our codes in a tree structure, where the codewords represent the leaves of our tree. An example is shown below:



Above, in addition to the characters $\{a, b, c, d, e, f\}$, we've included *frequency information.* That is, $f(a) = 0.45$ means that the probability of a random character in this language being equal to a is .45.

The code for each character can be found by concatenating the bits of the path from the root to the leaves. By convention, every left branch is given the bit 0 and every right branch is given the bit 1.

As long as the characters are on the leaves of this tree, the corresponding code will be prefix-free. This is because one string is a prefix of another if and only if the node corresponding to the first is an ancestor of the node corresponding to the second. No leaf is an ancestor of any other leaf, so the code is prefix-free.

1.3.2 How good is a code?

Suppose we have a set of characters C with frequencies $f(c)$ so that $\sum_{c \in C} f(c) = 1$. That is, $f(c)$ can be thought of as the probability of using a letter c in this language. The *cost*, in terms of bits, of a character $c \in C$ when using the coding scheme represented by a tree T is just the depth in the tree T: $\text{cost}(c) = d_T(c)$. For example, in the tree above, e has depth 4 in the tree, and requires 4 bits to represent. The average cost of the tree is

$$B(T) = \mathbb{E}_{c \in C}[d_T(c)] = \sum_{c \in C} f(c) d_T(c).$$

We say that a tree T is *optimal* if this expected cost $B(T)$ is as small as possible.

1.3.3 Huffman Codes

In 1951, David A. Huffman, in his MIT information theory class, was given the choice of a term paper or final exam. Huffman chose to do the term paper rather than take the final exam. He found greedy algorithm to find the most efficient binary code, which we know today as **Huffman codes**.

The basic idea is this: build subtrees for subsets of characters and merge them from the bottom up, combining the two trees with the characters of minimum total frequency.

Algorithm 2: A high-level version of the Huffman Coding algorithm.

```
Input: Set of characters $C = \{c_1, c_2, \dots, c_n\}$ of size $n$, and
  $F = \{f(c_1), f(c_2), \dots, f(c_n)\}$, a set of frequencies.
Create nodes $N_k$ for each character $c_k$, with key $f(c_k)$.
Let current denote the set $\{N_1, \dots, N_n\}$ of nodes.
while current has length more than one do
    Find the two nodes $N_i$ and $N_j$ in current with the minimum frequencies and create
      a new intermediate node $I$ with $N_i$ and $N_j$ as its children, so that
      $I$.key $= N_i$.key $+ N_j$.key.
    Add $I$ to current and remove $N_i$, $N_j$.
return the only entry of current, which is the root of the tree.
```

The tree shown above results from running this algorithm on the letters with those frequencies; see the slides for an illustration of this process.

1.3.4 Proof of Correctness

This algorithm works, but at first it's not at all obvious why. For a rigorous proof, refer to Lemmas 16.2 and 16.3 in CLRS. However, we'll sketch the idea below. Formally, the proof goes by induction. Recall that after iteration t in Algorithm 2, we have a list `current`, which contains the roots of subtrees that we still need to merge up. We will maintain the following inductive hypothesis:

- **Inductive hypothesis:** Suppose we have completed t iterations of the loop in Algorithm 2. Then there exists a way to merge the subtrees in `current` that is optimal.
- For the **base case**, we observe that when $t = 0$, `current` is just the set of all characters, and definitionally there exists an optimal tree made out of these nodes.
- For the **inductive step,** we need to show that if the inductive hypothesis holds at step $t - 1$, then it holds at step t. We'll sketch this later.
- Finally, to **conclude** the argument, we see that at the end of the algorithm, there is only one element in `current`, and in this case the inductive hypothesis reads that there is a

way to merge this single subtree to obtain an optimal subtree. That's just a convoluted way of saying that the single tree we return is optimal, and so we are done.

All that remains to show is the inductive step. We first observe the following claim:

Proposition 2. *We are given a set of characters C and a set of its associated frequencies F where $f(c)$ is the frequency of character c. Let x and y be the characters with the two smallest frequencies. There exists an optimal coding tree for C such that x, y are sibling leaves.*

Proof. Let T be the optimal coding tree for C. The optimal coding tree must be a full binary tree, that is, every non-leaf node must have two children. Let a, b be characters that are sibling leaves of maximum depth. We define the number of bits to encode c as $d_T(c)$ and the number of bits needed for the coding tree as $B(T) = \sum_c f(c)d_T(c)$.

We can replace a, b by x, y without increasing the total number of bits needed for the coding tree.[1] If we swap x and a, the change in cost becomes

$$f(x)d_T(a) + f(a)d_T(x) - f(x)d_T(x) - f(a)d_T(a) = (f(x) - f(a))(d_T(a) - d_T(x)) \leq 0$$

Therefore, swapping a, b with x, y will not increase our objective function $B(T)$. Hence, there exists an optimal coding tree where x, y are siblings in the tree. □

Proposition 2 shows that there exists an optimal coding tree where x and y are sibling leaves, that is, there is an optimal code that makes the same greedy choice as the algorithm. However, this is only immediately helpful for the first iteration of the inductive step, when all of the elements of `current` are indeed leaves. In order to make this idea work for all t, we need one more claim.

Proposition 3. *Let C be a set of characters, and let T be an optimal coding tree for C. Imagine creating C' from C by collapsing all the characters in a subtree rooted at a node N with key $k = N.key$ into a single character c' with frequency k. Then the corresponding tree T' is optimal for C'.*

Conversely, suppose that a tree T' that is an optimal coding tree for an alphabet C'. Let $c' \in C'$ be a character with frequency $f(c')$. Introduce new characters $c''_1, \ldots, c''_r$ with total frequency $\sum_{i=1}^{r} f(c''_i) = f(c')$. Let T'' be an optimal coding tree on $c''_1, \ldots, c''_r$. Then the tree T on the alphabet $C = (C' \setminus \{c'\}) \cup \{c''_1, \ldots, c''_r\}$ that has the leaf c' replaced with the subtree T'' is optimal.

Proof. Let T and T' be the two trees described in the lemma, and consider the difference of

[1] For simplicity, we ignore the case where a, b, x, y are not distinct. For more details, see Lemma 16.2 in CLRS.

their costs.

$$\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) \cdot d_T(c) - \sum_{c \in C'} f(c) \cdot d_{T'}(c) \\
&= \left(\sum_{i=1}^{r} f(c_i'') d_T(c_i'') \right) - f(c') d_{T'}(c') \\
&= \left(\sum_{i=1}^{r} f(c_i'') (d_{T''}(c_i'') + d_{T'}(c')) \right) - f(c') d_{T'}(c') \\
&= \sum_{i=1}^{r} f(c_i'') d_{T''}(c_i'') + d_{T'}(c') \sum_{i=1}^{r} f(c_i'') - f(c') d_{T'}(c') \\
&= \sum_{i=1}^{t} f(c_i'') d_{T''}(c_i'')
\end{aligned}$$

where the last line used the fact that $\sum_{i=1}^{r} f(c_i'') = f(c')$, and so the last two terms cancelled. This means that the difference in the cost between these two trees *only* depends on T'', it doesn't depend at all about the structure of T. Thus, T is optimal if and only if T' is optimal. □

The two Claims together prove the inductive step, because the second claim implies that the logic of the first claim holds, even for newly created intermediate nodes I.

Note: The proof in CLRS has the same basic steps (Lemmas 16.2 and 16.3 instead of the claims above), although phrased slightly differently. The sketch above is pretty sketchy, so if the above is hard to follow, please check out CLRS for a more detailed version.

CS 161 (Stanford, Winter 2022) Lecture 15

Adapted from Virginia Williams's lecture notes. Additional credits go to Logan Short, William Chen and Mary Wootters. Some of the figures in these notes are taken from CLRS.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

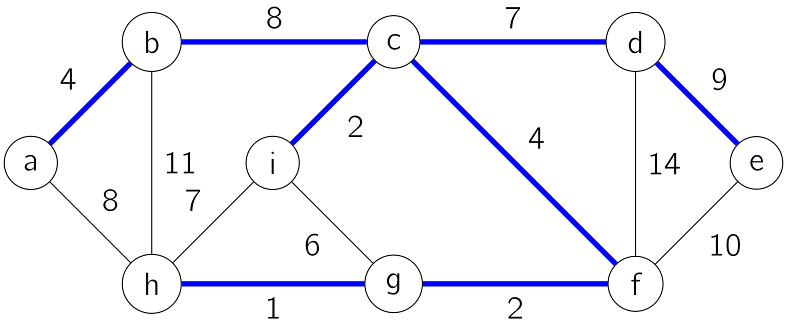
Minimum Spanning Trees

1 Introduction

Today we will continue our discussion of greedy algorithms, specifically in the context of computing minimum spanning trees. There are many useful applications for finding a minimum spanning tree of a graph from efficient network design to graph clustering analysis and much more. We will also show that we can compute a minimum spanning tree of a graph in polynomial time using some intuitive greedy algorithms.

The minimum spanning tree problem is formulated informally as follows: we are provided an undirected graph $G = (V, E)$ with weights $w(e) \in \mathbb{R}$ for $e \in E$ and we want to compute a subgraph of G that is a tree which connects all vertices in V (a spanning tree) and has minimum total edge weight defined as $w(T) = \sum_{e \in T} w(e)$.

Below is an example of an MST of a graph. In the example, the edges forming the MST are colored blue while edges that are not part of the MST are colored black:



2 A Template for Minimum Spanning Tree Algorithms

Let's start by introducing a basic algorithm template which will guide our discussion towards the actual algorithms for computing MSTs. These algorithms will in general follow the steps described in the template below:

We will show that the template results in a valid MST by maintaining the invariant that there exists at least one MST which contains all the edges in A. An edge is considered safe to add to A as long as it maintains this invariant. We will see that this definition of a safe edge

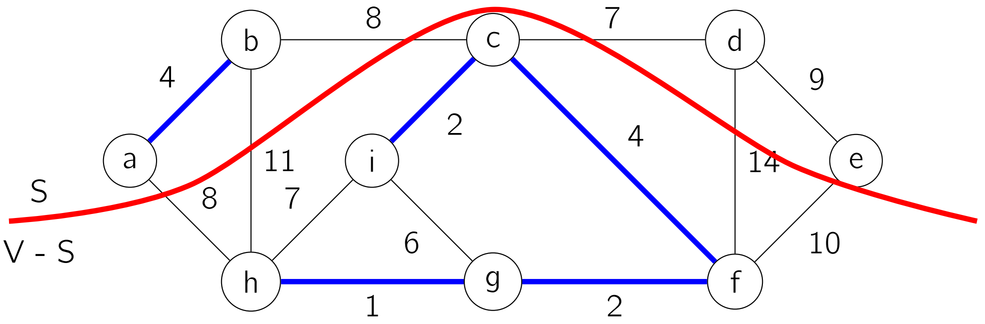
Algorithm 1: Template for Minimum Spanning Tree Algorithms

```
$A \leftarrow \emptyset$
while $A$ is not a spanning tree do
    find edge $(u, v)$ that is 'safe' for $A$
    $A \leftarrow A \cup \{u, v\}$
return $A$
```

can informally be defined as the edge with minimum weight which would not form a cycle if included in A. The next section will introduce new terminology to define this formally.

3 Cuts and Light Edges

We will introduce the notion of graph cuts to formally discuss which edges can be considered safe to add to the MST edge set. Let a *cut* $(S, V - S)$ of a graph $G = (V, E)$ be a partition of V into two disjoint sets S and $V - S$. From this, we can say that an edge (u, v) *crosses* the cut $(S, V - S)$ if the edge has one endpoint in S and the other in $V - S$. We can also say that a cut *respects* a subset A of edges if no edges in A cross the cut. An edge is considered a *light edge* crossing a cut if its weight is the minimum of any edge crossing the cut.

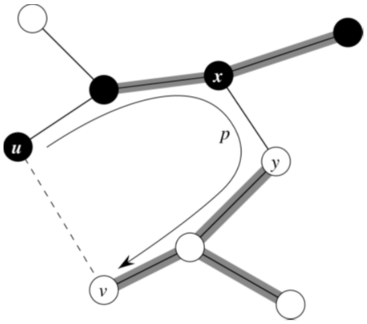


In the example above, let $(S, V - S)$ be a cut of the graph where S contains the set of nodes above the red curve and $V - S$ contains the set of nodes below it, and the set A be the set of edges colored blue. The edges which cross the cut are exactly the following: $(a, h), (b, h), (b, c), (c, d), (d, f), (e, f)$ and the only light edge which crosses $(S, V - S)$ is (c, d). Since none of these edges are contained in the set A, the cut respects the set A. Note that if we were to add any of the edges previously mentioned to A, then the cut would no longer respect A.

Given the definitions above, let $G = (V, E)$ be a connected and undirected graph with edge weights $w(e)$, A be a subset of E such that some MST of G contains A, $(S, V - S)$ be a cut that respects A, and (u, v) be a light edge crossing $(S, V - S)$.

Theorem 1. *There exists an MST that contains $A \cup \{(u, v)\}$.*

Proof. Let T be an MST containing A. As previously mentioned, (u, v) is a light edge which crosses the cut $(S, V - S)$. Since T is already a spanning tree, note that adding any other edge of the graph to it will lead to a cycle, so in particular adding (u, v) to T produces a cycle. Consider a path p from u to v in T. There will necessarily be at least one edge (x, y) of p which crosses the cut $(S, V - S)$ where $(x, y) \notin A$ because the cut respects A. Since (u, v) is a light edge, $w(u, v) \leq w(x, y)$. Deleting (x, y) from T and adding (u, v) yields a new MST T'. The only difference between T and T' are the edges (x, y) and (u, v) so $w(T') \leq w(T)$. T' is an MST which contains $A \cup \{(u, v)\}$. □



Note that in the proof, if $w(T') \neq w(T)$, then we have that our initial assumption of T being an MST is false since we have found a spanning tree with smaller total edge weight.

Because of the theorem, we can add some additional points about the MST algorithm template.

- The MST algorithm maintains a subset A of edges with no cycles. That is, the graph represented by $G_A = (V, A)$ is a forest (a set of distinct unconnected trees).
- Any safe edge (u, v) connects two distinct connected components of G_A.
- For some connected component $C = (V_C, E_C)$ in G_A, the safe edge (u, v) is a light edge crossing $(V_C, V - V_C)$.

4 Prim's Algorithm

At a high level, the set A maintained by Prim's algorithm is a single tree. The algorithm starts with an arbitrary root r and in each step, a light edge leading out of A and connecting to a node that has not yet been connected to A is selected and added to A. Once A connects every node in the graph, it is returned as an MST of the graph.

Prim's algorithm is similar to Dijkstra's algorithm in that estimates of the distance to each node are maintained and updated as the algorithm progresses. Q is a priority queue maintaining distances of vertices not in the tree so far, key(v) is the minimum weight of edge connecting v to some vertex in the tree, and p(v) is the parent of v in the tree.

Correctness Much of the correctness of Prim's algorithm follows from Theorem 1. Notice that at the beginning of every loop iteration, $A = \{\ (\text{p}(v), v) : v \in (V - \{r\} - Q)\}$ meaning that the vertices already placed in the partial MST are those in $V - Q$. For all vertices $v \in Q$,

Algorithm 2: Prim(G)

```
key($v$) ← $\infty$, $\forall v \in V$
key($r$) ← 0
$Q$ ← (key($v$), $v$), $\forall v \in V$
p($v$) ← NIL, $\forall v \in V$
$A \leftarrow \emptyset$
while $Q$ is not empty do
    $u$ ← ExtractMin($Q$)
    if $u \neq r$ then
        $A = A \cup \{(p(u), u)\}$
    for each neighbor $v$ of $u$ do
        if $v \in Q$ and $w(u, v) <$ key($v$) then
            key($v$) = $w(u, v)$
            DecreaseKey(key($v$), $v$)
            p($v$) = $u$
return $A$
```

if $p(v) \neq$ NIL, then key(v) is the minimum weight of an edge connecting v to the partial MST. This can be thought of in terms of graph cuts with partitions $(Q, V - Q)$ and the vertices in Q with non-NIL parents as being the tail of edges crossing this cut. Since in Q, only the vertices with non-NIL parents have key $\neq \infty$ (except for r in the first iteration), this means that only the edges which cross the cut are considered at each iteration and the one with minimum weight is added to A. This is exactly what the MST template algorithm does (we add a safe edge) and as such, the correctness of the algorithm follows.

Running time Prim's Algorithm can be implemented as a direct modification of Dijkstra's Algorithm and can achieve a similar running time, but its exact bound depends on the implementation of the priority queue.

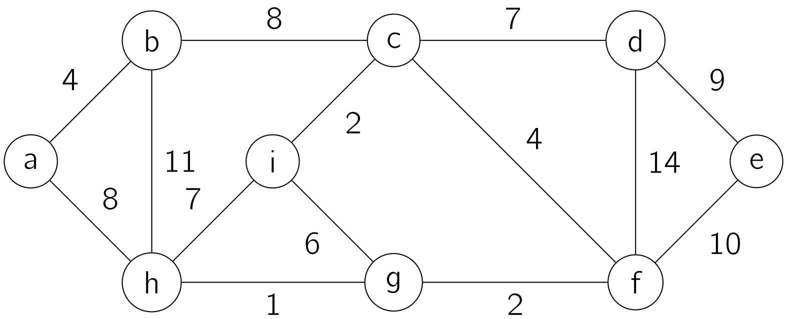
If a red-black tree or a binary heap is used:

- ExtractMin: $O(\log n)$
- DecreaseKey: $O(\log n)$
- Total: $O(n \log n + m \log n) = O(m \log n)$

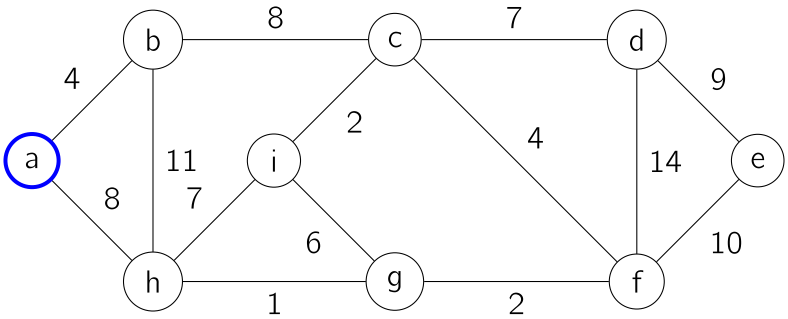
If a Fibonacci heap is used:

- ExtractMin: $O(\log n)$
- DecreaseKey: $O(1)$ amortized
- Total: $O(n \log n + m)$

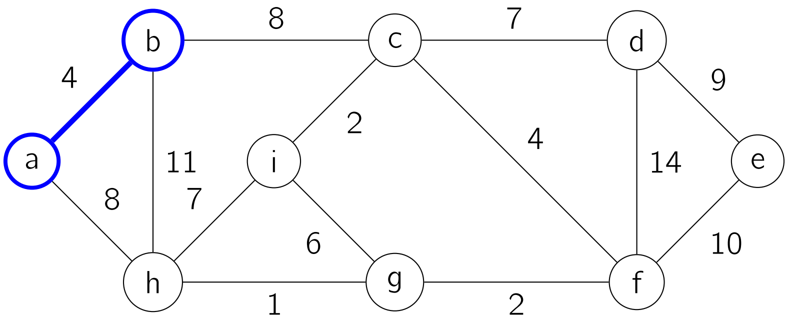
Example In this example we will run through the steps of Prim's algorithm in order to find an MST for the following graph:



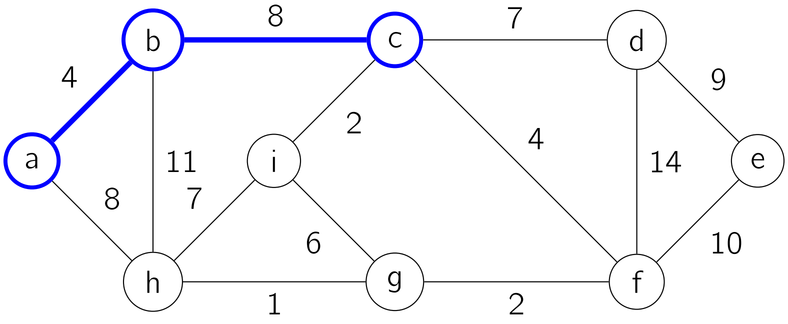
Suppose we select node a to be the source node, r. We then extract node a from Q and set key(b) = 4 , p(b) = a, key(h) = 8, and p(h) = a.



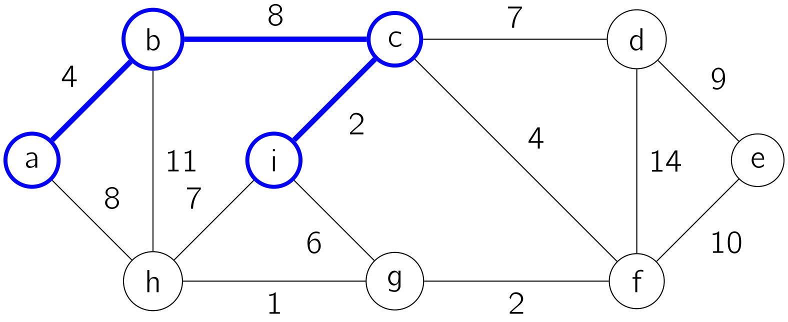
Since key(b) is now the smallest value in the priority queue, we visit node b. Because p(b) = a we add edge (a, b) to the set A. We then update the keys and parent fields of nodes that have edges connecting to b. Thus we set key(c) = 8 and p(c) = b.



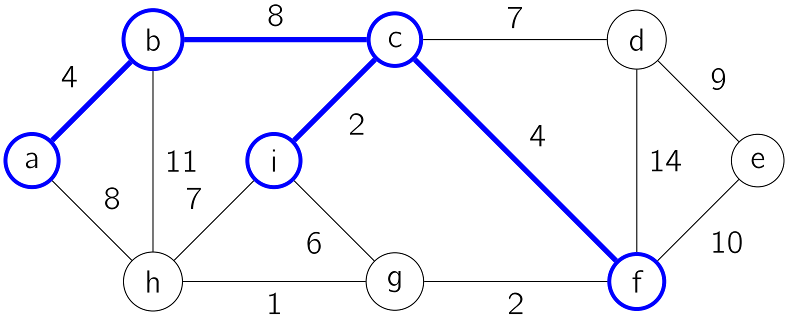
The next smallest in the priority queue is a tie between key(c) and key(h). The algorithm can pick either one – the results may be different, but both will be an MST. Let's say the algorithm arbitrarily picks c. We add edge (b, c) to A and perform the following updates: key(d) = 7, p(d) = c, key(f) = 4, p(f) = c, key(i) = 2, and p(i) = c.



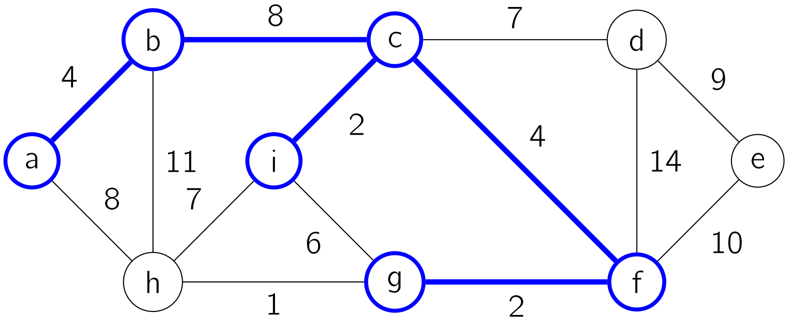
key(i) is the smallest so we visit node i. Update the following: key(g) = 6, p(g) = i, key(h) = 7, and p(h) = i.



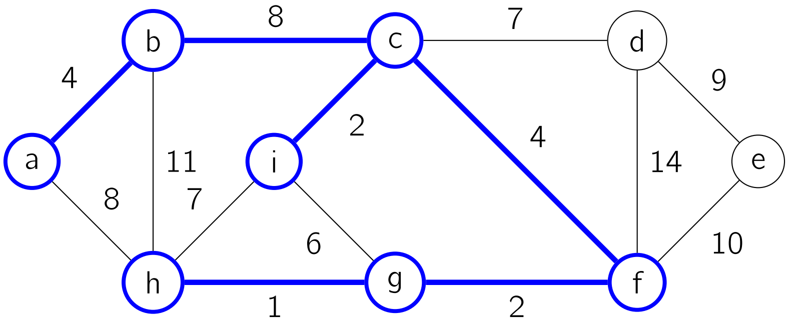
key(f) is the smallest so we visit node f. Update the following: key(g) = 2, p(g) = f, key(e) = 10, and p(e) = f.



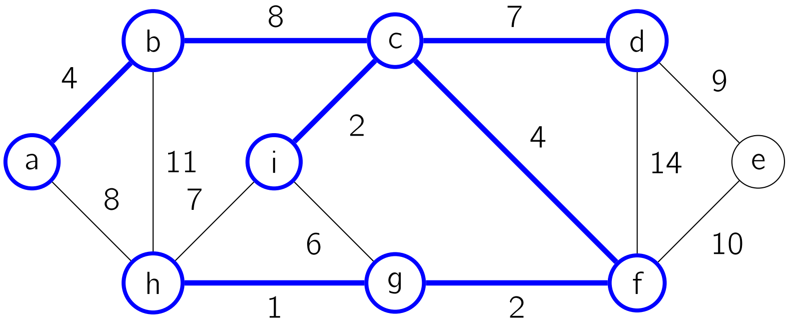
key(g) is the smallest so we visit node g. Update the following: key(h) = 1 and p(h) = g.



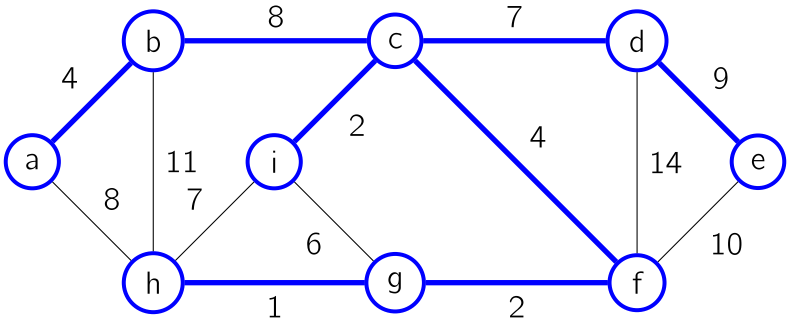
key(h) is the smallest so we visit node h. There are no updates at this step.



key(d) is the smallest so we visit node d. Update the following: key(e) = 9 and p(e) = d.



Finally, key(e) is the smallest so we visit node e. There are no updates at this step and the algorithm will detect that Q is empty at the next iteration and return.



5 Kruskal's Algorithm

At a high level, the set A maintained by Kruskal's algorithm is a set of disjoint trees. During update step i, if the ith smallest edge connects different trees, merge the two trees connected by this edge. The algorithm progresses until eventually only one tree remains at which point the set A represents an MST of the graph.

Kruskal's algorithm utilizes the union-find (aka disjoint set) data structure in order to handle the merging of the disjoint trees maintained by the algorithm. The union-find data structure supports disjoint sets with the following operations:

- **makeset(u):** creates a new set containing u provided that u is not in any other set
- **find(u):** returns the name of the set containing u
- **union(u, v):** merge the set containing u and the set containing v into one set

The algorithm itself can be structured as follows:

Algorithm 3: Kruskal(G)

```
$A \leftarrow \emptyset$
$E' \leftarrow$ sort edges by weight in non-decreasing order
foreach $v \in V$ do
    makeset($v$)
foreach $(u, v) \in E'$ do
    if $find(u) \neq find(v)$ then
        $A \leftarrow A \cup \{(u, v)\}$
        union($u, v$)
return $A$
```

Correctness The correctness follows from Theorem 1.

Running time The runtime of Kruskal's algorithm depends on two factors: the time to sort the edges by weight and the runtime of the union-find data structure operations. While $\Omega(m \log n)$ time is required for sorting the edges if we use comparison-based sorting, in many cases, we may be able to sort the edges in linear time. (Recall, that RadixSort can be used to sort the edges in $O(m)$ time if the weights are given by integers bounded by a polynomial in m.) In this case, the runtime is bounded by the runtime of the union-find operations and is given by $O(nT(\text{makeset}) + mT(\text{find}) + nT(\text{union}))$. The best known data structure supporting the union-find operations runs in amortized time $O(\alpha(n))$ where $\alpha(n)$ is the inverse Ackermann function. Interestingly, the value of the inverse Ackermann is tiny for all practical purposes:

$$\alpha(n) \leq 4, \quad \forall n < \texttt{\# atoms in the universe}$$

and thus for all practical purposes, the union-find operations run in constant time. Thus, in many settings, the runtime of Kruskal's algorithm is nearly linear in the number of edges.

The actual definition of $\alpha(n)$ is $\alpha(n) = \min\{k \mid A(k) \geq n\}$, where $A(k)$ is the Ackermann function evaluated at k. $A(k)$ itself is defined using the more general Ackermann function as $A(k) = A_k(2)$. $A_k(x)$ is defined recursively:

$$A_m(x) = \begin{cases} x + 1 & m = 0 \\ A_{m-1}(1) & m > 0, x = 0 \\ A_{m-1}(A_m(x-1)) & else \end{cases}$$

For example

- $A_0(x) = 1 + x$, so $A_0(2) = 3$.
- To compute $A_1(x)$, we see:

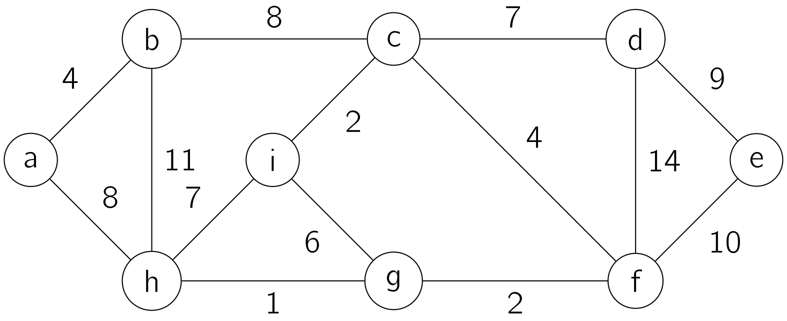
  $A_1(x) = A_0(A_1(x-1)) = A_0(A_0(A_1(x-2))) = \cdots = A_0(A_0(\cdots(A_0(A_1(0))))) = A_0(A_0(\cdots(A_0(2)))),$

  that is, we have "iterated" A_0 x times. If we work it out, we get

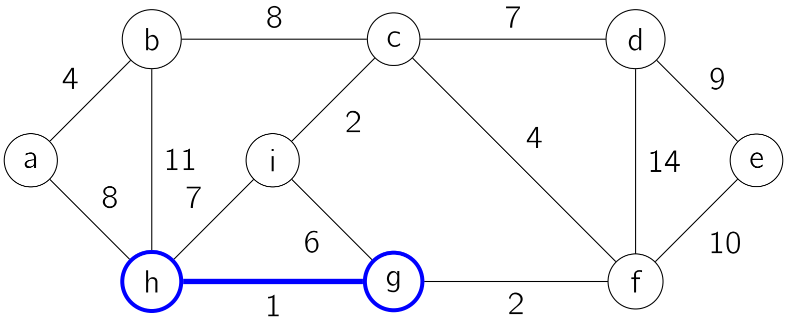
  $$A_1(x) = 2x.$$

  Thus, $A_1(2) = 4$.
- $A_2(x) = 2^x x$ (A_1 iterated x times), so $A_2(2) = 8$
- $A_3(x) \geq 2^{2^{2^{2\cdots}}}$, a "tower" of x 2s (A_2 iterated x times); it turns out $A_3(2) \geq 2^{11}$
- $A_4(x)$ is larger than the total number of atoms in the known universe, and also larger than the number of nanoseconds since the Big Bang. (Thus, $\alpha(n) \leq 4$ for all practical purposes.)

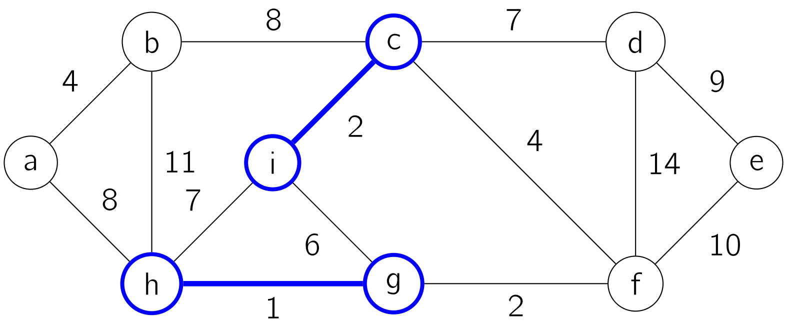
Example In this example we will run through the steps of Kruskal's algorithm in order to find an MST for the following graph:



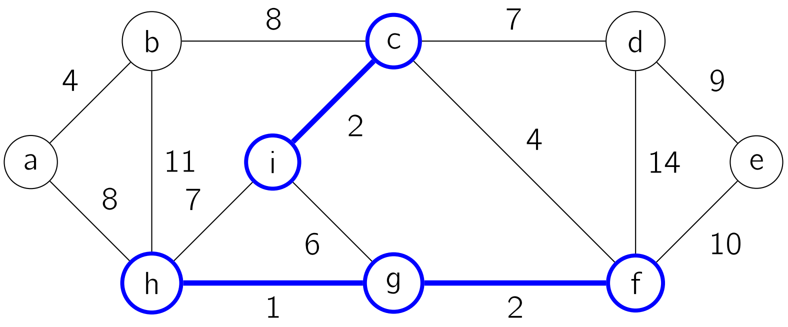
We begin by creating a new set for each node in the graph. We then begin iterating over the edges in non-decreasing order. The first edge we examine is (g, h). This edge connects nodes g and h which are currently not part of the same set. We thus include this edge in A and union the sets containing g and h.



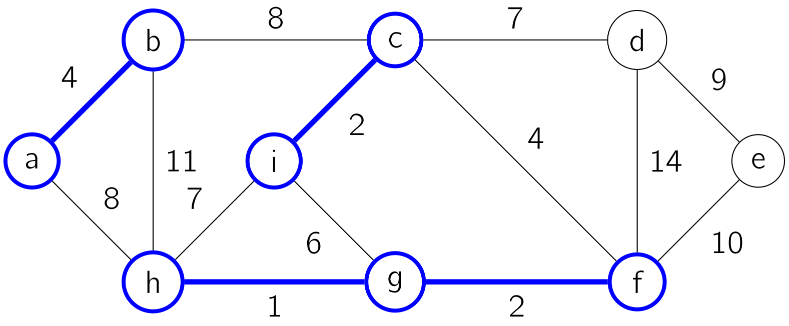
The next edge in the sorted order is a tie between edges (c, i) and (f, g). Picking either one will yield a correct result, so let's say the algorithm picks (c, i). Since c and i are not part of the same set we include this edge in A and union the sets containing c and i.



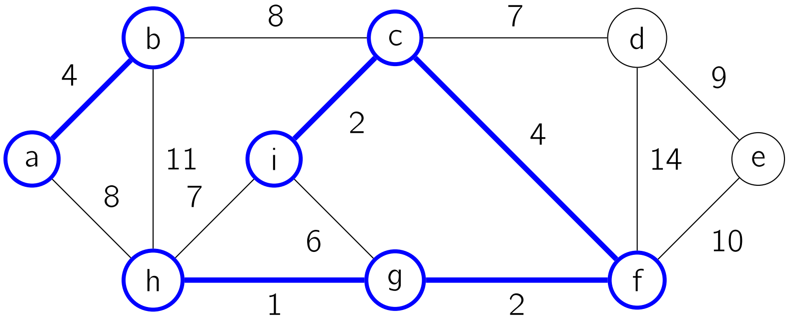
The next edge in the sorted order is (f, g). We union the sets containing f and g and add edge (f, g) to A.



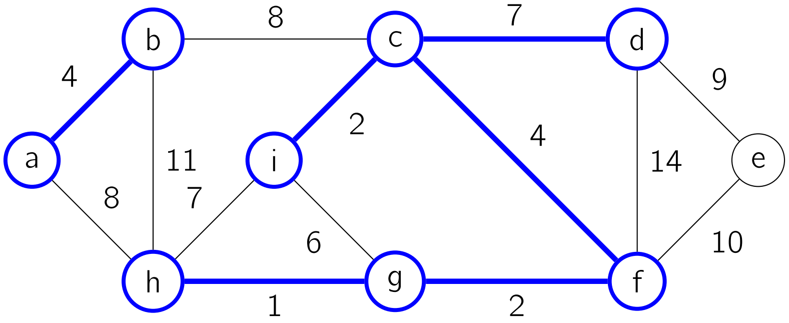
The next edge in the sorted order is a tie between edges (a, b) and (c, f). Let's say the algorithm picks (a, b). We union the sets containing a and b and add edge (a, b) to A.



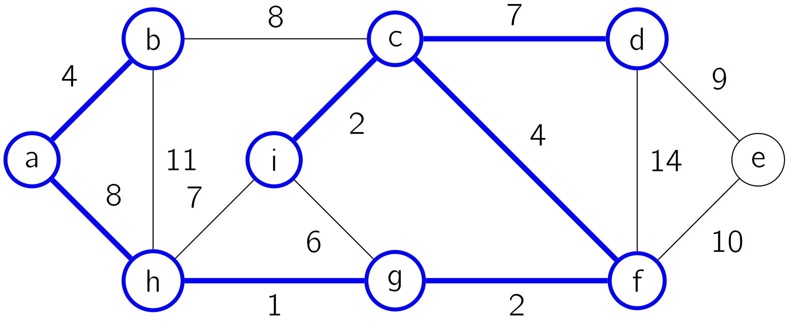
The next edge in the sorted order is (c, f). We union the sets containing c and f and add edge (c, f) to A.



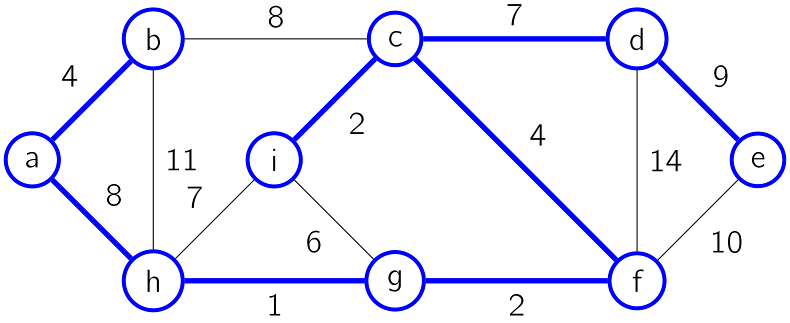
The next edge in the sorted order is (i, g). Note that i and g are contained in the same set which means that adding the edge (i, g) would lead to a cycle in A. We therefore ignore (i, g) and pick the next edge in sorted order which is a tie between (c, d) and (h, i). Let's say the algorithm picks (c, d). We union the sets containing c and d and add edge (c, d) to A.



The next edge in the sorted order is (h, i), but h and i are contained in the same set so we ignore it. The next edge is a tie between edges (a, h) and (b, c). Let's say the algorithm picks (a, h). We union the sets containing a and h and add edge (a, h) to A.



The next edge in the sorted order which has both vertices in different sets is (d, e). We union the sets containing d and e and add edge (d, e) to A. At this point all nodes are contained in the same set, so no further edges are added to A.



6 The Latest and Greatest Algorithms

While the greedy algorithms mentioned above are reasonably efficient ways to compute a minimum spanning tree of a graph, recent research has yielded more efficient algorithms. In 1995, Karger, Klein, and Tarjan discovered a randomized linear time ($O(E+V)$) algorithm based on Borůvka's algorithm and the reverse-delete algorithm. In 2000, Chazelle discovered the current fastest determistic algorithm which runs in time $O(E\alpha(V))$ using soft heaps where α is the inverse Ackermann function.

CS 161 (Stanford, Winter 2022) Lecture 16

Adapted from Virginia Williams' lecture notes. Additional credits go to Chuanqi Shen, Luke Johnston, Gregory Valiant, and Mary Wotters.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

Max Flow, Min Cut and Ford-Fulkerson

1 History of Flows and Cuts

Today we will study a very interesting problem called the max flow problem. Before we get to the algorithm and math, we briefly discuss the interesting history behind max flow! During the Cold War, the US Air Force at that time was very interested in the Soviet train networks. In reports that were declassified in 1999, it was revealed that the Air Force collected enough information about the train network that they were able to determine how resources were shipped from the Soviet Union to Europe. The Air Force was very interested in determining how much resources can be transported from the Soviet Union to Europe, and what needed to be done to destroy this movement of resources. What this translates to is the min cut problem, i.e., cutting the minimum number of train tracks so that nothing goes to Europe. Here, cutting an edge meant dropping a bomb. Nowadays, however, there are much more peaceful applications of this problem, for instance, understanding the flow of information through the Internet.

2 Formulation of the Maximum Flow Problem

You are given an input graph $G = (V, E)$, where the edges are directed. There is a function $c : E \to \mathbb{R}_{\geq 0}$ that defines the capacity of each edge. We also label two nodes in G, s and t, as the source and destination, respectively. The task is to output a *flow* of *maximum value*. We will shortly define what a *flow* is and what a flow of *maximum value* means.

A flow f is a function $f : E \to \mathbb{R}_{\geq 0}$ such that

1. Capacity constraints are satisfied:

$$\forall (u, v) \in E : \quad 0 \leq f(u, v) \leq c(u, v)$$

2. Flow conservation constraints are satisfied:

$$\forall v \in V - \{s, t\} : \quad \sum_{x \in N_{\text{in}}(v)} f(x, v) = \sum_{y \in N_{\text{out}}(v)} f(v, y)$$

Here $N_{\text{in}}(v)$ denotes the set of nodes with an edge that points to v and $N_{\text{out}}(v)$ denotes the set of nodes that v points to.

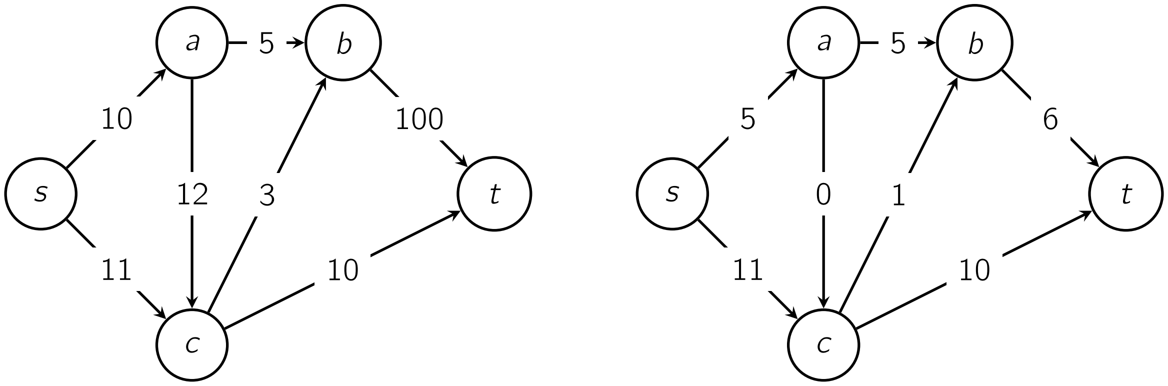


Figure 1: (Left) Graph G with edge capacities (Right) Graph G with a sample flow.

Suppose that there are no edges going into s and no edges coming out of t. From the above, you can verify yourself that $\sum_{x \in N_{\text{out}}(s)} f(s,x) = \sum_{y \in N_{\text{in}}(t)} f(y,t)$. We define the value $\sum_{x \in N_{\text{out}}(s)} f(s,x)$ to be the value of the flow f. We usually denote the value of a flow f as $|f|$. If there are edges going into s and out of t, then the value of f is

$$|f| = \sum_{x \in N_{\text{out}}(s)} f(s,x) - \sum_{y \in N_{\text{in}}(s)} f(y,s).$$

The max flow problem is to find some flow f such that $|f|$ is maximized.

Remark 1. In the analysis below we consider graphs with a single source s and a single sink t. However, if we need to work with a graph with multiple sources, we can do so by adding a new source node, and then adding edges with capacity infinity from it to each of the multiple sources. Similarly, if we want to have multiple sinks, we add a new sink node and add edges from the multiple sinks to that sink with capacity infinity.

3 Example

In Fig. 1, we have a graph G and a sample flow f. Observe that the two constraints for a flow are satisfied. There can be multiple other flows possible that can satisfy the constraints. For our given flow, $|f| = 16$. The max flow for this graph is actually 18, as we will see shortly.

4 Formulation of the Minimum Cut Problem

Now, we give a formulation of the min cut problem defined for directed graphs with source and destination nodes s and t. Note that there is also a version of the min cut problem without a source and sink node, though we won't discuss that now.

An s-t cut is a partition $V = S \cup T$ where S and T are disjoint and $s \in S, t \in T$, and the size/cost of an s-t cut is

$$c(S,T) := \sum_{x \in S, y \in T} c(x,y).$$

For our graph G shown above, if we set $S = \{s, a, c\}$ and $T = \{b, t\}$, then the cost of the cut is $c(a, b) + c(c, b) + c(c, t) = 5 + 3 + 10 = 18$. If we take another cut $S' = \{s, c\}, T' = \{a, b, t\}$, then $c(S', T') = c(s, a) + c(c, b) + c(c, t) = 10 + 3 + 10 = 23$. Note that we *do not* consider the edge $\{a, c\}$ as it is in the wrong direction (we only consider edges from S' to T').

5 The Max-Flow Min-Cut Theorem

Lemma 2. *For any flow f and any s-t cut (S, T) of G, we have $|f| \le c(S, T)$. In particular, the value of the max flow is at most the value of the min cut.*

Proof.

$$\begin{aligned}
|f| &= \sum_{x \in N_{\text{out}}(s)} f(s, x) - \sum_{y \in N_{\text{in}}(s)} f(y, s) \\
&= \sum_{v \in S} \left(\sum_{x \in N_{\text{out}}(v)} f(v, x) - \sum_{y \in N_{\text{in}}(v)} f(y, v) \right) \text{ [by flow conservation constraint for } v \neq s] \\
&= \sum_{v \in S} \left(\sum_{x \in N_{\text{out}}(v) \cap S} f(v, x) - \sum_{y \in N_{\text{in}}(v) \cap S} f(y, v) \right) + \sum_{v \in S} \left(\sum_{x \in N_{\text{out}}(v) \cap T} f(v, x) - \sum_{y \in N_{\text{in}}(v) \cap T} f(y, v) \right) \\
&= \sum_{v \in S} \left(\sum_{x \in N_{\text{out}}(v) \cap T} f(v, x) - \sum_{y \in N_{\text{in}}(v) \cap T} f(y, v) \right) \text{ [first term sums to 0]} \\
&\le \sum_{v \in S, x \in T, x \in N_{\text{out}}(v)} f(v, x) \le \sum_{v \in S, x \in T, x \in N_{\text{out}}(v)} c(v, x) = c(S, T).
\end{aligned}$$

In the proof, $\sum_{v \in S} \left(\sum_{x \in N_{\text{out}}(v) \cap S} f(v, x) - \sum_{y \in N_{\text{in}}(v) \cap S} f(y, v) \right) = 0$ since we add and subtract the flow $f(u, v)$ for every $u, v \in S$ such that $(u, v) \in E$. □

We get the following consequence.

Corollary 3. *If we can find f and (S, T) such that $|f| = c(S, T)$, then f is a max flow and (S, T) is a min s-t cut.*

It turns out that we can always find such f and (S, T) for any graph.

Theorem 4 (Max-Flow Min-Cut Theorem)**.** *For any graph G, source s and destination t, the value of the max flow is equal to the cost of the min cut.*

We will show this by coming up with an algorithm. The algorithm will take the graph G and some flow f that has already been constructed, and create a new graph that is called the residual graph. In this new graph, the algorithm will try to find a path from s to t. If no such path exists, we will show that the value of the flow we started with is the value of the

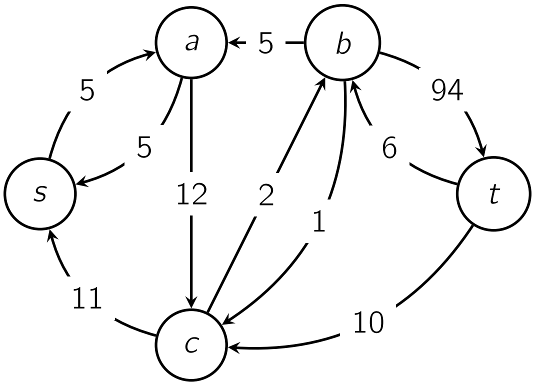


Figure 2: The residual network given the flow presented in Fig. 1.

maximum flow. If not, we show how to increase the value of our flow by pushing some flow on that path.

6 The Ford-Fulkerson Max-Flow Algorithm

We will make an assumption on our graph. The assumption can be removed, but it will make our lives easier. We will assume that for all $u, v \in V$, G does not have both edges (u, v) and (v, u) in E. We can make this condition hold by modifying the original graph in the following way. If $(u, v), (v, u) \in E$, we split the edge (u, v) to two edges (u, x) and (x, v), where x is a new node we introduce into the graph. This makes the number of nodes at most $m + n$.

Now, let f be a flow given to us. We will try to see if we can improve this flow. We will define the *residual capacity* $c_f : V \times V \to \mathbb{R}_{\geq 0}$ as follows.

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E, \\ f(v, u) & \text{if } (v, u) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Basically, what this does is that, if there is any flow through the edge, you remove the flow from the capacity and add an edge in the opposite direction with the value of the flow. The reason we do this is because the flow we picked thus far might not be the correct flow, and this formulation allows us to undo changes that we have done through "adding flow" in the revere direction. The residual capacity $c_f(u, v)$ represents how much flow we can send from u to v in addition to the flow f.

We define G_f to be a residual network defined with respect to f, where $V(G_f) = V(G)$ and $(u, v) \in E(G_f)$ if $c_f(u, v) > 0$. Figure 2 shows G with the residual edges.

We will show that, if there is a path from s to t in G_f, then f is not a max flow. If no such path exists, that f is a max flow.

Lemma 5. *If t is not reachable from s is G_f, then f is a maximum flow.*

Proof. Let S be the set of nodes reachable from s in G_f and $T = V - S$. There are no edges in G_f from S to T since the nodes in T are not reachable from s. Note that (S, T) defines an s-t cut. Now consider any $v \in S, w \in T$. We have $c_f(v, w) = 0$ since (v, w) is not an edge in G_f. There are three cases:

1. If $(v, w) \in E$, then by definition $c_f(v, w) = c(v, w) - f(v, w) = 0 \implies c(v, w) = f(v, w)$.
2. If $(w, v) \in E$, then $c_f(v, w) = f(w, v) = 0$.
3. If $(v, w) \notin E$ and $(w, v) \notin E$, we can disregard (v, w) and (w, v) since they do not appear in any flow or cut.

Using this, and the proof in Lemma 2, we have

$$\begin{aligned}
|f| &= \sum_{v \in S} \left(\sum_{x \in N_{\text{out}}(v) \cap T} f(v, x) - \sum_{y \in N_{\text{in}}(v) \cap T} f(y, v) \right) \\
&= \sum_{v \in S} \sum_{x \in N_{\text{out}}(v) \cap T} f(v, x) \text{ [from case 2 the second term is 0]} \\
&= \sum_{v \in S} \sum_{x \in N_{\text{out}}(v) \cap T} c(v, x) \text{ [from case 1]} \\
&= c(S, T).
\end{aligned}$$

Thus, we show that the flow is equal to the cut. From Corollary 3 we know that f is a maximum flow, and (S, T) is a min cut. □

Lemma 6. *If G_f has a path from s to t, we can modify f to f' such that $|f| < |f'|$.*

Proof. Pick a path P from s to t in G_f, and consider the edge of minimum capacity on the path. Let that capacity be F. Then we can increase our flow by F. For each edge in P, if $c_f(v, w)$ is the right direction (i.e., there is an edge $(v, w) \in E(G)$), then we can increase our flow on this edge by F. If $c_f(v, w)$ is in the opposite direction (i.e., $(w, v) \in E(G)$), then we can decrease the flow on this edge by F. In effect, we are "undoing" the flow on this edge. By doing so, we have increased our flow by F.

As an example, consider Fig. 2 again. The path $s \to a \to c \to b \to t$ is a path with minimum capacity 2. Therefore, we can update our flow and push additional 2 units of flow, resulting in a flow of 18.

Formally, Let $s = x_0 \to x_1 \to \ldots \to x_k = t$ be a simple path P in G_f, and let $F = \min_i c_f(x_i, x_{i+1})$. Define a new flow f' where

$$f'(u, v) = \begin{cases} f(u, v) + F & \text{if } (u, v) \in P, \\ f(u, v) - F & \text{if } (v, u) \in P, \\ f(u, v) & \text{otherwise.} \end{cases}$$

We now need to show that f' is a flow. The capacity constraints are satisfied because for every $(u, v) \in E$,

1. If $(u, v) \in P$, then $0 \leq f(u, v) + F \leq f(u, v) + c_f(u, v) = f(u, v) + c(u, v) - f(u, v) = c(u, v)$.
2. If $(v, u) \in P$, then $f(u, v) - F \leq f(u, v) \leq c(u, v)$ and $f(u, v) - F \geq f(u, v) - c_f(v, u) = 0$.
3. Otherwise, $f(u, v)$ is from the original flow f.

The conservation constraints are also satisfied: Recall that P is a simple path. Thus, for every $v \in V - \{s, t\}$, P uses 0 or two edges incident on v. If P uses 0 edges on v, then flow values of the edges incident on v have not changed when going from f to f'. Thus, suppose that P uses two edges (x, v) and (v, y) incident on v. Because in G_f some edges appear in the opposite direction compared to G, we need to consider a few cases.

1. (x, v) and (v, y) are both in the same direction (an edge into v and an edge out of v); the flow into v increases by F and the flow out of it also increases by F.
2. (x, v) and (v, y) are both in the opposite direction ($(v, x), (y, v) \in E$); the flow into v decreases by F and the flow out of it also decreases by F.
3. (x, v) is in the correct direction and (v, y) is in the opposite direction. Then the flow into V changes by $F - F = 0$.
4. (x, v) is in the opposite direction and (v, y) is in the correct direction. Then the flow out of V changes by $F - F = 0$.

Finally, note that we increase our flow by F. Consider the edge (s, x_1) in P. If $(s, x_1) \in E$, the flow out of s increases by F. If $(x_1, s) \in E$, the flow into s decreases by F. By our definition of G_f, it must be that $F > 0$, and we get that $|f'| > |f|$. □

From this, we can construct an algorithm to find the maximum flow. Starting with some arbitrary flow of the graph, construct the residual network, and check if there is a path from s to t. If there is a path, update the flow, construct the new residual graph and repeat. Otherwise, we have found the max flow.

A path from s to t in the residual graph is called an *augmenting* path, and pushing flow through it to modify the current flow is referred to as *augmenting* along the path.

The run time of this algorithm is bounded by the number of times we update our flow. If the edge capacities are all integers, we can increase the flow by at least 1 each time we update our flow. Therefore, the runtime is $O(|f|m)$ where $|f|$ is the value of the max flow. If we have rational edge capacities, then we can multiply all edge capacities by a factor to make them all integers. However, the runtime blows up by that factor as well. If we have irrational edge capacities, then the algorithm is no longer guaranteed to terminate. So we have a problem.

Algorithm 1: maxflow(G, s, t)

```
$f \leftarrow$ all zeroes flow
$G_f \leftarrow G$
while $t$ is reachable from $s$ in $G_f$ (check using, e.g., DFS or BFS) do
    $P \leftarrow$ path in $G_f$ from $s$ to $t$
    $F \leftarrow$ min capacity on $P$
    $f \leftarrow f'$ as defined in Lemma 6
    Update $G_f$ to correspond to new flow
return $f$
```

We will save the day in the next sections. Algorithm 1 is called the Ford-Fulkerson method. It is actually part of a family of algorithms that depend on how the path P between s and t in G_f is selected. One can obtain P via DFS, BFS, or any other method for selecting paths. It turns out that two methods work particularly well: the shortest path method and the fattest path method. The shortest path method is known as the Edmonds-Karp algorithm or Dinic's algorithm.

The fattest path method. This method finds a path between s and t that maximizes $\min_{e \in P} c_f(e)$ among all s-t paths P. Finding such a path can be done in $O(m+n)$ time by a clever mix of linear time median-finding and DFS.

The shortest path method (the Edmonds-Karp algorithm/Dinic's algorithm). This method picks the path between s and t using BFS, thus picking a path that minimizes the number of edges. Finding such a path also runs in $O(m)$ time: BFS takes $O(m+n)$ to explore the whole graph, but since we only care about the vertices reachable from s this is $O(m)$ time.

Since both methods of selecting a path run in linear time, the main question becomes, how many iterations does Ford-Fulkerson perform? We will answer these questions below in the next section.

7 Running time of various implementations of Ford-Fulkerson

Remark 7. We did not discuss the details of this section in class, but it's in the notes for the interested reader.

7.1 The fattest path version of Ford-Fulkerson

In this section we will show that the fattest path method results in a runtime of $O(m(m+n)\log|f|)$ when run on a graph with integer capacities. Thus, when rational capacities are converted to integers by multiplying by N, we get a runtime of $O(m(m+n)(\log|f|+\log N))$

for rational capacities. Thus the effect of large N is mitigated. This method does not solve the issues when the capacities can be irrational.

To show the runtime, we prove a main claim that states that after each iteration of the algorithm, the maximum flow value in G_f goes down by a factor of $(1-1/m)$. This max flow value starts as $|f|$ since $G_f = G$ in the beginning of the algorithm, and ends at 0 as in the end s and t are disconnected.

Proposition 8 (Main)**.** *Let f' be the max flow in G_f. Then after one iteration of Ford-Fulkerson on G_f, the max flow value becomes $\leq |f'|(1-1/m)$..*

Proof. Let P be the fattest path from s to t in G_f. Let $F = \min_{e\in P} c_f(e)$. Let S be the nodes reachable from s in G_f via paths composed of edges with residual capacities $> F$. Thus, any edge (x, y) of G_f with $x \in S$, $y \notin S$ must have $c_f(x, y) \leq F$. In particular, this means that the size of the cut between S and $V\S$ is $\sum_{x\in S, y\in V\setminus S} c_f(x, y) \leq mF$. Thus, the size of the min s-t cut in G_f is at most mF.

By the max-flow-min-cut theorem from last lecture, the size of the min s-t cut is at least the size of the max-flow $|f'|$ in G_f, and so $|f'| \leq mF$. Thus $F \geq |f'|/m$.

Now, when we augment (push flow) along P, the flow in G increases by F, while the flow in G_f decreases by F. Thus, the new flow in G_f after augmenting along P becomes $|f'| - F \leq |f'|(1 - 1/m)$. □

Now that the main claim has been proven, we can conclude with a discussion of the runtime. Consider how the max flow value in G_f evolves after t iterations. It starts as $|f|$ (where f is the max flow in G) and then after t iterations is

$$\leq |f|\,(1-1/m)^t\,.$$

If $t = m \ln |f|$, we get that the max flow value in G_f is

$$\leq |f|\,((1-1/m)^m)^{\ln|f|} < |f|(1/e)^{\ln|f|} = 1.$$

Since all the capacities are integers, all the residual capacities are also integers, and so the max flow value in G_f is an integer. Since it is < 1, it must be 0. Hence after $m \ln |f|$ iterations, the max flow value in G_f is zero, s and t are disconnected and the computed flow in G is maximum. The runtime is $O((m + n)m \log |f|)$.

7.2 The shortest path version of Ford-Fulkerson

Here we analyze running Ford-Fulkerson using BFS to find a path between s and t in G_f.

With each augmentation along a path P in G_f, at least one edge is removed from G_f, namely the edge with residual capacity $F = \min_{e\in P} c_f(e)$. The main claim that we need to prove the runtime is that the number of times an edge can be removed from G_f is small. Since each iteration of the algorithm causes at least one removal, the main proposition will show that the number of iterations is small and hence the runtime is small as well.

Proposition 9 (Main)**.** *Fix any (u, v) that is ever an edge in G_f. Then the number of times that (u, v) can disappear from G_f is at most $n/2$.*

Once this proposition is proven, we would get that the total number of edge disappearances is at most $mn/2$ and hence the number of iterations of the algorithm is also $\leq mn/2$. Because of this, the algorithm's runtime is $O((m + n)mn)$.

To prove the proposition, we will need a useful lemma (see below) that shows that as G_f evolves through the iterations, for any v, the (unweighted) distance from s to v in G_f cannot go down. Let's begin with some notation. Let G_f^i be the residual network after the i-th iteration of the algorithm; $G_f^0 = G$. For a vertex v, let $d_i(v)$ be the (unweighted) distance from s to v in G_f^i.

Lemma 10. *For all $i \geq 1$, and all $v \in V$, $d_{i-1}(v) \leq d_i(v)$.*

Proof. Fix i. We will prove the statement for i by induction on $d = d_i(v)$.

The inductive hypothesis is that for all d and all v with $d_i(v) = d$, $d_{i-1}(v) \leq d_i(v)$. The base case is $d = 0$. We note that if $d_i(v) = 0$, then $v = s$ since we view G_f^i as an unweighted graph. But then we also have $d_{i-1}(s) = 0 \leq d_i(s)$.

For the induction, let's assume that the inductive hypothesis holds for $d - 1$, i.e. that for all x with $d_i(x) = d - 1$, $d_{i-1}(x) \leq d_i(x)$. We want to show that for all v with $d_i(v) = d$, we also have $d_{i-1}(v) \leq d_i(v)$.

Consider some v with $d_i(v) = d$. Let u be the node just before v on a shortest $s - v$ path in G_f^i. Then, $d_i(u) = d_i(v) - 1 = d - 1$ and the inductive hypothesis applies to it so that $d_{i-1}(u) \leq d_i(u)$.

We consider two cases.

Case 1. $(u, v) \in G_f^{i-1}$. Then, by the triangle inequality in G_f^{i-1}, we have that $d_{i-1}(v) \leq d_{i-1}(u) + 1$. Since $d_{i-1}(u) \leq d_i(u)$, we get that

$$d_{i-1}(v) \leq d_i(u) + 1 = (d_i(v) - 1) + 1 = d_i(v).$$

Case 2. $(u, v) \notin G_f^{i-1}$. Then, since $(u, v) \in G_f^i$, we must have that (v, u) was on the $(i - 1)$st augmenting path. Hence $d_{i-1}(u) = d_{i-1}(v) + 1$. Hence:

$$d_{i-1}(v) = d_{i-1}(u) - 1 \leq d_i(u) - 1 = d_i(v) - 2 \leq d_i(v).$$

In both cases $d_{i-1}(v) \leq d_i(v)$ and the induction is complete. □

Now we are ready to prove the main proposition.

Fix some (u, v) that is an edge in G_f at some point. Let's consider two consecutive disappearances of (u, v). Suppose that $(u, v) \in G_i$ but $(u, v) \notin G_{i+1}$. If after this disappearance

(u, v) had another one later on, then at some point (u, v) must have appeared in G_f again. Let j be the first iteration after i so that the jth augmenting path made (u, v) appear in G_f^{j+1}.

Because $(u, v) \in G_f^i$ but $(u, v) \notin G_f^{i+1}$, (u, v) must have been in the i-th augmenting path P_i. Because $(u, v) \notin G_f^j$ but $(u, v) \in G_f^{j+1}$, (v, u) must have been in the j-th augmenting path P_j.

From this we obtain that $d_i(v) = d_i(u) + 1$ and $d_j(u) = d_j(v) + 1$. Using the fact that $j > i$ and the key lemma from above we obtain

$$d_j(u) = d_j(v) + 1 \geq d_i(v) + 1 = d_i(u) + 2.$$

Thus, between (u, v)'s disappearance and its next reappearance, the distance from s to u increased by $+2$. Hence between any two consecutive disappearances the distance to u increases by ≥ 2. The distance starts as ≥ 0 and can be $\leq n - 1$ before becoming ∞. Thus the total number of disappearances of (u, v) is $\leq n/2$. This completes the proof of the main claim and the proof of the runtime.

8 Applications

We wrap up by talking about some applications of the Ford-Fulkerson algorithm.

8.1 Bipartite perfect matching

Let $G = (V, E)$ be an undirected, unweighted *bipartite graph*: the set of vertices is partitioned into V_1 and V_2 so that there are no edges with two endpoints entirely in V_1 or entirely in V_2. A matching in G is a collection of edges, no two of which share an end point. A perfect matching is a matching M such that every node in V has exactly one incident edge in M. In order for G to have a perfect matching, we need that $|V_1| = |V_2|$. The perfect matching problem is, given a bipartite graph G with $|V_1| = |V_2| = n$ and on m edges, determine whether G has a perfect matching.

We will solve the bipartite perfect matching problem by creating an instance of max flow and using Ford-Fulkerson's algorithm.

Given $G = (V_1 \cup V_2, E)$, direct all the edges in E from V_1 to V_2. Add two extra nodes s and t. Add (directed) edges from s to every node in V_1 and from every node of V_2 to t. In this new graph H, let all the edge capacities be 1 and then run the Ford-Fulkerson algorithm to compute the max flow.

Suppose that G has a perfect matching M. Then, H has max flow value $n = |V_1| = |V_2|$. This is because we can set $f(e) = 1$ for every $e \in M$, ell the edges out of s and all the edges out of t. All other flow values are 0. The capacity constraints are trivially satisfied. The flow conservation constraints are satisfied since for every $x \in V_1$ there is exactly one edge (s, x) into x that has flow 1, and exactly one edge $(x, y) \in M$ with flow 1; similarly for every $x \in V_2$

there is exactly one edge (x, t) out of x that has flow 1, and exactly one edge $(y, x) \in M$ with flow 1.

Suppose now that Ford-Fulkerson returns a flow f of value n. Hence $f(s, x) = f(y, t) = 1$ for all $x \in V_1, y \in V_2$. Because Ford-Fulkerson causes all flow values on the edges to be integers, the flow values on all edges are either 1 or 0. Because of this, every node $x \in V_1$ gets flow of 1 going into it and a flow of 1 needs to come out so that there is a single edge (x, y) that has flow value 1 and all other edges out of x have flow value 0. Similarly, for every $y \in V_2$ there is a unique edge into y with positive flow value 1. The edges in $V_1 \cup V_2$ with positive flow through them must hence form a perfect matching.

8.2 More applications

There are many applications of max-flow and min-cut! We may talk about a few more in class if time (check the slides).

CS 161 (Stanford, Winter 2022) Lecture 17

Adapted from lecture notes by Aviad Rubinstein and modified by Nima Anari.
Please direct all typos and mistakes to Moses Charikar and Nima Anari.

Stable Matching and Gale-Shapley

Every year, tens of thousands of doctors begin their residencies in the United States. The process of matching doctors to their residencies is extremely complex, and must take into account the doctors' preferences and the needs of the hospitals. In this lecture we will learn the **Deferred Acceptance Algorithm** which is at the core of *The Match*, a.k.a. the National Resident Matching Program. The algorithm is sometimes also called the *Gale-Shapley* algorithm after its inventors, David Gale and Lloyd Shapley.

1 Stable matching

Setting: Our goal is to match a collection of doctors and a collection of hospitals to each other. Throughout, we assume that each hospital has exactly one position to fill. This is just to simplify the verbiage and notation, and does not lose generality; in the more general case where hospitals have larger capacity, replace the word "hospital" by "hospital position" in the ensuing text and everything still goes through. We also assume the number of hospitals and doctors is the same and we call that n. This is again to simplify the notation; when these numbers are not equal, some doctors (or hospitals) might have to remain unmatched. We can model this by adding fake hospitals (or doctors) that are ranked very low, and matching to them counts as being "unmatched" in real life.

Suppose that we had some numerical score for each pair of doctor i and hospital j hinting at how good of a fit pairing i and j is. Then, we may try to design an algorithm that finds an optimal (say max-weight) matching in this graph.

One problem with this approach is that we do not have numerical scores, so we have to rely on information from doctors and hospitals themselves. We might ask each doctor to rank the hospitals, and each hospital to rank the doctors. But even then, it is not straightforward how to translate the preferences of both a doctor and a hospital, two rankings, into a single numerical score. Any such scheme can also be prone to "gaming" by the doctors and the hospitals, because they can possibly manipulate the outcome and gain a better match by reporting false preferences.

Even worse, in order to use a centralized matching algorithm, you must convince thousands of residency programs to list their positions on your algorithm and commit to taking the doctors you assign them. So you better ensure that hospitals can't find better matches outside your algorithm!

This motivates the definition of a stable matching:

Definition 1 (Stable Matching)**.** We say that a bipartite matching between doctors and hospitals is *stable* if for every doctor i and hospital j, one of the following happens:

- i, j are matched to each other; or
- Hospital j is matched to another doctor which it prefers over i; or
- Doctor i is matched to another hospital which it prefers over j.

Intuitively, if doctor i prefers Hospital j to its match in the algorithm, and hospital j prefers doctor i, then they would jointly prefer to find their match without your algorithm. Such a pair (i, j) is called a *blocking pair*, and a matching is stable if it does not admit any blocking pairs.

So finding a stable matching resolves the problem of doctors and hospitals trying to bypass your algorithm. (There is still the issue of misrepresenting preferences. We will get back to this issue in Section 4.)

Example 2. Consider three doctors {Alice, Bob, Charlie}, and three hospitals $\{X, Y, Z\}$.

Suppose that Alice prefers Y over X over Z, etc:

Table 1: Doctor preferences

Doctor	Alice	Bob	Charlie
top choice	Y	X	X
2nd choice	X	Y	Y
last choice	Z	Z	Z

On the hospital side, their preferences are:

Table 2: Hospital preferences

Hospital	X	Y	Z
top choice	Alice	Charlie	Bob
2nd choice	Charlie	Alice	Charlie
last choice	Bob	Bob	Alice

- Observe that the matching

$$\text{(Alice-X),(Bob-Z),(Charlie-Y)} \tag{1}$$

  is stable: every hospital gets its favorite doctor, so no hospital wants to hire a doctor outside the matching. (The doctors are less happy about it...)

- Another stable matching is

$$\text{(Alice-Y),(Bob-Z),(Charlie-X).} \tag{2}$$

- The matching

$$\text{(Alice-Z),(Bob-X),(Charlie-Y)} \tag{3}$$

is *not* stable, because Alice and X prefer each other over their respective matches.

2 The algorithm

The **Deferred Acceptance Algorithm** is essentially based on a greedy strategy: at each step we try to match each doctor to her most preferred hospital, except for hospitals that we already have tried and know are impossible.

There is a twist which distinguishes this algorithm from other greedy algorithms we've seen previously in the course. The algorithm's greedy choices are revocable, i.e., the matches are not final until the end of the algorithm. Even if, for example, doctor i matched with hospital j at the first round of the algorithm, if in a later iteration, doctor k also wants to match with hospital j, we will break ties between the doctors based on hospital j's preferences. In particular, if hospital j prefers k over i, then the previous match between j and i is canceled, and doctor i is sent to find a new hospital.

Algorithm 1: Gale-Shapley, a.k.a. the Deferred Acceptance Algorithm

```
input  : $n \times n$ matrices $D$ and $H$. The $i$-th row of $D$ is the list of hospitals in the order
         of most preferred to least preferred from doctor $i$'s view. $H[j][i]$ is the rank of
         doctor $i$ from hospital $j$'s perspective (lower number means more preferred).
output : Matching between doctors and hospitals.
                                                   // Initialization
for $d \in doctors$ do
    $d.i \leftarrow 0$
for $h \in hospitals$ do
    $h$.doctor $\leftarrow$ NIL
freeDoctors $\leftarrow$ doctors
                          // While there is a free doctor we keep matching.
while $freeDoctors \neq \emptyset$ do
    Pick any $d \in$ freeDoctors
    // $h$ is the next preferred hospital for $d$.
    $d.i \leftarrow d.i + 1$
    $h \leftarrow D[d][d.i]$
                            // If $h$ prefers $d$ over previous match, switch.
    if $H[h][d] < H[h][h.doctor]$ then
        Add $h$.doctor to freeDoctors
        Remove $d$ from freeDoctors
        $h$.doctor $\leftarrow d$
return the matching $\{(h.doctor, h) \mid hospitals\ h\}$
```

3 Proof of correctness

In this section, we will prove that Algorithm 1 always outputs a stable matching. Let us start by proving some observations about the behavior of Algorithm 1.

Proposition 3. *If h.doctor becomes not NIL, it will never become NIL again.*

Proof. We only assign NIL values at initialization. So once a hospital has a *potential match* it can never run out of matches again; it will only reject a potential match for another match (a *more preferred* one). □

Proposition 4. *In Algorithm 1, no doctor "runs out of" hospitals to try. In other words $d.i$ never becomes $n+1$.*

Proposition 5. *If a doctor d were to run out of hospitals to try, it means that they must have already tried all hospitals. But this means every hospital already has a potential match; at the time the hospital rejected d, they must have already had another match, and by the previous proposition, the hospital's match never becomes NIL from that point onwards.*

But now we get a contradiction. There are n hospitals, each of which has a current match other than d. So all of the current hospital matches must come from the set of $n-1$ doctors other than d, which is impossible.

Proposition 6. *The algorithm terminates in at most $O(n^2)$ iterations and returns a matching.*

Proof. In every iteration, some $d.i$ gets incremented by 1. Since they start at 0 and can never reach $n+1$, in total the number of increments is at most $n \times O(n) = O(n^2)$. By the previous proposition, when the algorithm terminates, we have a full matching. □

Now that we know Algorithm 1 terminates and returns a matching, it only remains to prove that the matching is **stable**.

Theorem 7. *The matching returned by Algorithm 1 is stable.*

Proof. First note that a hospital's matches only improve over the course of the algorithm. In other words, for every h, h.doctor's ranking according to h can only get better and better as we run through the algorithm. This is because h will only reject a potential match for a better one.

Let us call the matching returned by Algorithm 1, σ. Assume by contradiction, that there is a blocking pair (i, j) for σ. This means that i prefers j to her match in σ and j prefers i to its match in σ. Let h be the hospital that i got matched to in σ. Since doctors go over their preference list one-by-one in Algorithm 1 and h comes later in the list of i than j, this means at some point during the run of the algorithm j must have been "proposed to" by i. So j had the opportunity to keep i or potentially reject i for better and better matches. But j ended up with a match that's worse than i; this is a contradiction, since j's matches must have only gotten better throughout the algorithm. □

4 Doctor-optimality and incentive compatibility

Let σ be a stable matching. We say that it is *doctor-optimal* if for any other stable matching σ' and any doctor d, the doctor d weakly prefers her hospital in σ to her hospital in σ'. (*Weakly prefer* means that it is either better or the same.) Note that a priori it is not clear that doctor-optimal stable matchings even exist.

In Example 2, the doctor-optimal matching is given in Eq. (2), and Eq. (1) is the stable matching that hospitals prefer. This is not a coincidence: the doctor-optimal stable matching is provably always the hospital-worst stable matching.

Theorem 8 (Doctor-optimality)**.** *The output of Algorithm 1 is* doctor-optimal*.*

Proof. For each doctor d, let $h^*(d)$ be her best feasible hospital, i.e,. the one she prefers the most among any hospital she can be matched to in any stable matching. Assume *by contradiction* that Algorithm 1 is not doctor-optimal. This means some doctor d gets rejected by her best possible choice $h^*(d)$ at some point during the algorithm. Consider the first time during the run of Algorithm 1 where this happens. This means $h^*(d)$ rejects d because a new doctor d' tries to match with it and $h^*(d)$ prefers d' over d. Now there are two cases:

- $h^*(d') = h^* = h^*(d)$, i.e., they both have the same best feasible hospital h^*. Therefore Hospital h^* will always prefer d' over d, and d' always prefers it to its match in any stable matching. Therefore, it is never stable to match h^* to d (d' and h^* would be a blocking pair). But this is a contradiction to h^* being feasible for d!
- $h^*(d') \neq h^*(d)$, in which case doctor d' should have first tried to match with $h^*(d')$ before coming to $h^*(d)$. If d' tries to match with $h^*(d)$ instead, it must have been rejected by $h^*(d')$. But this is a contradiction to d being the first doctor to be rejected by her best feasible hospital!

□

Incentive compatibility. Consider a doctor that strategically misreports her preferences in hope of a better hospital. Intuitively, given the new (misreported) preferences, Algorithm 1 should still output a stable matching. By the doctor-optimality theorem that we just proved, the doctor is already assigned the best match she can possibly be assigned in any stable matching; hence misreporting preferences cannot help her. This logic is actually flawed (can you see why?[1]), but the conclusion is still true, namely no doctor has an incentive to misreport her preferences:

Theorem 9 (Incentive-compatibility)**.** *Misreporting the preference over hospitals can never improve a doctor's match computed by Algorithm 1.*

[1] Given the misreported preferences, Algorithm 1 should still output a matching that is stable *with respect to the misreported preferences*. But the doctor-optimality theorem only guaranteed that the output is doctor-optimal among all matchings that are stable with respect to the true preferences.

The correct proof of this theorem is beyond the scope of these notes. It can be found, e.g., in this research paper by Dubins and Freedman: `https://www-jstor-org.stanford.idm.oclc.org/stable/2321753`.

As a remark, unlike doctors, hospitals can actually misreport and potentially get better matches. To appreciate the incentive compatibility for the doctors, let us revisit Example 2, and see how hospitals can strategically misreport their preferences to improve their matching.

Example 10. Suppose hospital X misreports its preferences as in Table 3 (but doctors' preferences are still the same).

Table 3: Hospital mis-reported preferences

Hospital	X	Y	Z
top choice	Alice	Charlie	Bob
2nd choice	**Bob**	Alice	Charlie
last choice	**Charlie**	Bob	Alice

Now Eq. (1) is the unique stable matching, so the Deferred Acceptance Algorithm will choose it, and not Eq. (2) (can be verified by running the algorithm).