ALGORITHMS IN PYTHON

Contents

Chapter 1

Basic Vocabulary

1.1 What is an algorithm?

An algorithm is a rote procedure for accomplishing a task (*i.e.*, a recipe). Some very algorithmic tasks include

- Cooking
- Knitting / weaving
- Sorting a deck of playing cards
- Shuffling a deck of playing cards
- Playing a song from sheet music / guitar tab
- Searching an $n \times n$ pixel image for a smaller $k \times k$ image (*e.g.*, of a face)

1.2 Classifications of algorithms

Since the notion of an algorithm is so general, it can be useful to group algorithms into classes.

1.2.1 Online vs. offline

Online algorithms are suitable for dynamically changing data, while offline algorithms are only suitable for data that is static and known in advance.

For example, some text editors can only perform "spell check" in an offline fashion; they wait until you request a spelling check and then process the entire file while you wait. Some more advanced text editors can perform spell check online. This means that as you type, spell check will be performed in realtime, *e.g.*, underlining a mispelled word like "asdfk" with a little red squiggle the moment you finish typing it. Checking the entire file every time you type a keystroke will likely be too inefficient in practice, and so an online implementation has to be more clever than the offline implementation.

Alternatively, consider sorting a list. An offline sorting algorithm will simply re-sort the entire list from scratch, while an online algorithm may keep the entire list sorted (in algorithms terminology, the sorted order of the list is an "invariant", meaning we will never allow that to change), and would insert all new elements into the sorted order (inserting an item into a sorted list is substantially easier than re-sorting the entire list).

Online algorithms are more difficult to construct, but are suitable for a larger variety of problems. There are also some circumstances where online algorithms are not currently possible. For example, in internet search, it would be very difficult to instantly update the database of webpages containing a keyword "asdfk" instantly upon someone typing the last keystroke of "asdfk" on their webpage.

1.2.2 Deterministic vs. random

Deterministic algorithms will perform identical steps each time they are run on the same inputs. Randomized algorithms on the other hand, will not necessarily do so.

Interestingly, randomized algorithms can actually be constructed to always produce identical results to a deterministic algorithm. For example, if you randomly shuffle a deck of cards until they are sorted, the final result will be the same as directly sorting the cards using a deterministic method, but the steps performed to get there will be different. (Also, we expect that the randomized algorithm would be much more inefficient for large problems.)

Randomized algorithms are sometimes quite efficient and can sometimes be made to be robust against a malicious user who would want to provide the

inputs in a manner that will produce poor performance. Because the precise steps that will be performed by randomized algorithms are more difficult to anticipate, then constructing such a malicious input is more difficult (and hence we can expect that malicious input may only some fraction of the time based on randomness).

For example, if you have a sorting algorithm that is usually fast, but is slow if the input list is given in reverse-sorted order, then a randomized algorithm would first shuffle the input list to protect against the possibility that a malicious user had given us the list in reverse-sorted order. The total number of ways to shuffle n unique elements is $n!$, and only one of them would produce poor results, and so by randomiznig our initial algorithm, we achieve a probability of $\frac{1}{n!}$ that the performance will be poor (regardless of whether or not our algorithm is tested by a malicious user). Since $n!$ grows so quickly with n, this means a poor outcome would be quite improbable on a large problem.

1.2.3 Exact vs. approximate

Exact algorithms produce the precise solution, guaranteed. Approximate algorithms on the other hand, are proven only to get close to the exact solution. An ϵ-approximation of some algorithm will not be guaranteed to produce an exact solution, but it is guaranteed to get within a factor of ϵ of the precise solution. For example, if the precise solution had value x, then an ϵ-approximation algorithm is guaranteed to return a value $\in [\ell(x, \epsilon), u(x, \epsilon)]$, where w.l.o.g. u can be defined as $x + \epsilon$, ϵx, or as some function of x, ϵ, and n that implies a useful bound.[1] Sometimes approximations use a hard-coded ϵ, *e.g.* a 2-approximation, but other times ϵ-approximations include ϵ in the runtime function, revealing how the runtime would change in response to a higher-quality or lower-quality approximation.

[1]Depending on the type of problem, this bound may be one-sided. For example, a 2-approximation of the traveling salesman problem should return a result $\in [x, 2x]$ where x is the optimal solution. This is because x is defined as the shortest possible tour length, and so in this case an estimate that falls below the best possible would likely be seen as unacceptable. In other contexts, a two-sided approximation would be fine; in those cases a 2-approximation would return a result $[\frac{x}{2}, 2x]$, where x is the correct solution. As a numeric bound, this may be interesting, but if a valid path is actually provided, it cannot possibly be better than the best path, and so something is wrong.

1.2.4 Greedy

Greedy algorithms operate by taking the largest "step" toward the solution possible in the short-term. For example, if you are a cashier giving someone change at a café, and the amount to give back is \$6.83, then a greedy approach would be to find the largest note or coin in American currency not larger than the remaining change:

1. \$5 is the largest denomination $\leq$ \$6.83; \$6.83 - \$5 = \$1.83 remaining.
2. \$1 is the largest denomination $\leq$ \$1.83; \$1.83 - \$1 = \$0.83 remaining.
3. 50 is the largest denomination $\leq$ \$0.83; \$0.83 - \$0.5 = \$0.33 remaining.
4. 25 is the largest denomination $\leq$ \$0.33; \$0.33 - \$0.25 = \$0.08 remaining.
5. 5 is the largest denomination $\leq$ \$0.08; \$0.08 - \$0.05 = \$0.03 remaining.
6. 1 is the largest denomination $\leq$ \$0.03; \$0.03 - \$0.01 = \$0.02 remaining.
7. 1 is the largest denomination $\leq$ \$0.02; \$0.02 - \$0.01 = \$0.01 remaining.
8. 1 is the largest denomination $\leq$ \$0.01; \$0.01 - \$0.01 = \$0.00 remaining.

Thus you give the change using a single \$5 note, a single \$1 note, a 50 cent piece, a quarter, a nickel, and three pennies.

Some problems will be solved optimally by a greedy algorithm, while others will not be. In American currency, greedy "change making" as above is known to give the proper change in the fewest notes and coins possible; however, in other currency systems, this is not necessarily the case. For example, if a currency system had notes of value 1, 8, and 14, then reaching value 16 would be chosen by a greedy algorithm as $16 = 1 \times 14 + 2 \times 1$, using 3 notes; however $16 = 2 \times 8$ would achieve this using only 2 notes.

If you've ever not crossed at a crosswalk in the direction you want to go during a green light because you know that this small instant gratification gain will slow you down at subsequent crosswalks (*e.g.*, crosswalks that do not have a stoplight, making you wait for traffic), you understand the hazard of greedy algorithms.

Listing 1.1: Recursive factorial.

```
def factorial(n):
  if n <= 1:
      # Note that factorial is only defined for non-negative values of
      # n
      return 1
  # factorial(n) is decomposed into factorial(n-1)
  return n*factorial(n-1)
```

1.2.5 Optimal vs. heuristic

An optimal algorithm is guaranteed to produce the optimal solution. For example, if you are searching for the best way to arrange n friends in a row of seats at the cinema (where each friend has a ranking of people they would most like to sit next to), then an optimal algorithm will guarantee that it returns the best arrangement (or one of the best arrangements in the case that multiple arrangements tie one another).

A heuristic algorithm on the other hand, is something that is constructed as "good in practice" but is not proven to be optimal. Heuristics are generally not favored in pure algorithms studies, but they are often quite useful in practice and are used in situations where practical performance on small or moderately sized problems matters more than theoretical performance on very large problems.

1.2.6 Recursive

Recursive algorithms decompose a problem into subproblems, some of which are problems of the same type. For example, a common recursive way to implement factorial in Python is shown in Listing 1.1.

Recursive methods must implement a base case, a non-recursive solution (usually applied to small problem sizes) so that the recursion doesn't become infinite.

1.2.7 Divide-and-conquer

Divide-and-conquer algorithms are types of recursive algorithms that decompose a problem into multiple smaller problems of the same type. They are

heavily favored in historically important algorithmic advances. Prominent examples of divide-and-conquer algorithms include merge sort, Strassen matrix multiplication, and fast Fourier transform (FFT).

1.2.8 Brute force

Brute force is a staple approach for solving problems: it simply solves a problem by trying all possible solutions. For example, you can use brute force to find all possible ways to make change of a $0.33 (Listing 1.2). The output of `all_ways_to_make_change(0.33)` is

```
solution:
3 x 0.01
1 x 0.05
1 x 0.25

solution:
8 x 0.01
1 x 0.25

solution:
8 x 0.01
5 x 0.05

solution:
13 x 0.01
4 x 0.05

solution:
18 x 0.01
3 x 0.05

solution:
23 x 0.01
2 x 0.05

solution:
28 x 0.01
1 x 0.05
```

Listing 1.2: Brute force for enumerating all possible ways to make change.

```
import numpy as np
import itertools

def all_ways_to_make_change(amount):
    # American currency:
    all_denominations = [0.01, 0.05, 0.25, 0.5, 1, 2, 5, 10, 20, 50, 100]
    # E.g., the maximum possible number of nickels will be amount / 0.05:
    possible_counts_for_each_denomination = [
        np.arange(int(np.ceil(amount/coin_or_note))) for coin_or_note in
        all_denominations ]

    for config in
        itertools.product(*possible_counts_for_each_denomination):
        total = sum([num*val for num,val in zip(config,all_denominations)])
        # if the coins actually add up to the goal amount:
        if total == amount:
            print 'solution:'
            for num,val in zip(config,all_denominations):
               # only print when actually using this coin or note:
               if num > 0:
                  print num, 'x', val
            print
```

1.2.9 Branch and bound

Closely related to brute force algorithms, branch-and-bound is a method for avoiding checking solutions that can be proven to be sub-optimal. Rather than try all permutations as performed in Listing 1.2, you could instead code a recursive algorithm that would abort investigating a potential solution once the subtotal exceeded the goal amount. For example, Listing 1.2 tries multiple solutions that simultaneously include both 33 pennies and 3 nickels (*e.g.*, 33 pennies, 3 nickels, 1 dime or 33 pennies, 3 nickels, 2 dimes, ...); all of these configurations can be aborted because they already exceed the goal amount (and because American currency does not feature negative amounts).

"Branch and bound" refers to a combination of brute force ("branch"ing in a recursive tree of all possible solutions) and aborting subtrees where all included solutions must be invalid or suboptimal ("bound"ing by cutting the recursive tree at such proveably poor solutions).

Chapter 2

Time and Space Complexity Bounds

2.1 Algorithmic Time and Space Bounds

The time and space used by an algorithm can often be written symbolically in terms of the problem size (and other parameters). This is referred to as the "time and space complexity" of the algorithm.

For example, a particular sorting algorithm may take $3n^2 + n$ steps and require $n+2$ space (*i.e.*, space to store $n+2$ values of the same type as those being sorted). In the analysis of algorithms, the emphasis is generally on the magnitude of the runtime and space requirements. For this reason, we are not generally interested in distinguishing between an algorithm that takes $3n^2$ steps and another that takes n^2 steps; $3n^2$ is slower than n^2, but only by a constant factor of 3. Such constant factor speedups are often implementation specific anyway. In one CPU's assembly language, an operation might take 3 clock cycles, while on other CPUs, it might take 1 clock cycle.

Likewise, we are not particularly interested in distinguishing $n^2 + n$ from the faster n^2; $n^2 + n$ is slower than n^2, but asymptotically they have identical performance:

$$\lim_{n \to \infty} \frac{n^2 + n}{n^2} = 1.$$

2.2 Big-oh Notation

"Big-oh" notation allows us to class time (or space) requirements by using the asymptotic complexity and by ignoring runtime constants (which would likely vary between implementations anyway). $O(n^2)$ is the set of all functions that can be bounded above by n^2 for $n > N$ (for some constant N) and allowing for some runtime constant C.

More generally,

$$O(f(n)) = \{g(n) : \exists N \ \exists C \ \forall n > N, \ g(n) < C \cdot f(n)\}.$$

We can see that

$$\begin{aligned} n^2 &\in O(n^2) \\ 3n^2 &\in O(n^2) \\ n^2 + n &\in O(n^2). \end{aligned}$$

Note that $n^2 \in O(n^2)$ and $O(n^2) \subset O(n^3)$, so $n^2 \in O(n^3)$ as well.

It is helpful to recognize why we need both N and C in the definition of $O(f(n))$. N is the point at which $f(n)$ becomes a ceiling for some $g(n) \in O(f(n))$. Of course, if we choose a large enough n, then a faster-growing $f(n)$ should start to dwarf $g(n)$. Since we are concerned only with asymptotic behavior, we are unconcerned with the fact that, *e.g.*, $100n > n^2$ when $n = 1$. So why might we need a constant C as well? Consider the fact that we want $10n^2 \in O(n^2)$; asmyptotically, these functions are within a constant factor of one another (*i.e.*, $\lim_{n\to\infty} \frac{10n^2}{n^2} = 10$); therefore, we could use $N = 1$ and $C = 10$ to show that $10n^2 \in O(n^2)$.

To get the tightest upper bound big-oh, we simply find the asymptotically most powerful term of the runtime function $f(n)$ and then strip away all constants:

$$\begin{aligned} & O(2n^2 \log(2n^2)) \\ = \ & O(4n^2 \log(2n)) \\ = \ & O(n^2 \log(2n)) \\ = \ & O(n^2(\log(n) + 2)) \\ = \ & O(n^2 \log(n)). \end{aligned}$$

2.2.1 Why don't we specify the log base?

When you see $O(\log(n))$, it is not uncommon to wonder what base logarithm we are performing: base-10 log, natural log (*i.e.*, base e), base 2 log, *etc.*; let us consider a log using an arbitrary constant a as the base. We see that we can transform this into a constant C multiplied with a logarithm from any other base b:

$$\begin{aligned} \log_a(n) &= \frac{\log_b(n)}{\log_b(a)} \\ &= C \cdot \log_b(n) \\ &= \in O(\log_b(n)). \end{aligned}$$

Thus, we see that all logarithms are equivalent when using $O(\cdot)$; therefore, we generally do not bother specifying the base.

2.3 Big-Ω Notation

Where big-oh (*i.e.*, $O(\cdot)$) provides an upper bound of the runtime, big Ω is the lower bound of the runtime:

$$\Omega(f(n)) = \{g(n) : \exists N \ \exists C \ \forall n > N, \ g(n) > C \cdot f(n)\}.$$

Clearly $n^2 \in \Omega(n^2)$ and $\Omega(n^2) \subset \Omega(n)$, and so $n^2 \in \Omega(n)$ as well.

2.4 Big-Θ Notation

If a function $f(n) \in O(g(n))$ and also $f(n) \in \Omega(g(n))$, then we say that $f(n) \in \Theta(g(n))$, meaning $g(n)$ is a tight bound for $f(n)$. That is to say, $g(n)$ asymptotically bounds $f(n)$ both above and below by no more than a constant in either direction.

For example, if we prove an algorithm will never run longer than $2n\log(n) + n$ and we also prove that the same algorithm can never be faster than $n\log(n^2)$, then we have

$$\begin{aligned} f(n) &\in O(n\log(n)) \\ f(n) &\in \Omega(n\log(n)) \\ \rightarrow f(n) &\in \Theta(n\log(n)); \end{aligned}$$

however, if only we prove that an algorithm will never run longer than $n\log(n)$ and we also prove that the same algorithm can never be faster than $2n$, then we have

$$\begin{aligned} f(n) &\in O(n\log(n)) \\ f(n) &\in \Omega(n) \end{aligned}$$

and we cannot make a tight bound with $\Theta(\cdot)$.

2.5 Little-oh and Little-ω

There are also $o(\cdot)$ and $\omega(\cdot)$, which correspond to more powerful versions of the statements made by $O(\cdot)$ and $\Omega(\cdot)$, respectively:

$$o(f(n)) = \left\{ g(n) : \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0 \right\}$$

$$\omega(f(n)) = \left\{ g(n) : \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \right\}.$$

For example, $n^2 \in o(n^2\log(n))$, meaning $n^2\log(n)$ must become an infinitely loose upper bound as n becomes infinite[1]. Likewise, $n^2\log(n) \in \omega(n^2)$, meaning n^2 must become an infinitely loose lower bound as n becomes infinite.

Although $n^2 \in O(n^2)$, $n^2 \notin o(n^2)$.

2.6 ϵ notation

$O(n^{2+\epsilon})$ denotes the set of all functions that are bounded above (when $n \gg 1$ and allowing for some constant C) by $O(n^{2+\epsilon})$ for any $\epsilon > 0$. For example, $n^2\log(n) \in O(n^{2+\epsilon})$. This is because, for any $\epsilon > 0$,

$$\begin{aligned} \lim_{n\to\infty} \frac{\log(n)}{n^\epsilon} &= \lim_{n\to\infty} \frac{\frac{\partial}{\partial n}\log(n)}{\frac{\partial}{\partial n} n^\epsilon} \\ &= \frac{\frac{1}{n}}{\epsilon \cdot n^{\epsilon-1}} \\ &= \frac{1}{\epsilon \cdot n^\epsilon}. \end{aligned}$$

[1] Note that we mean "infinitely loose" in terms of the ratio, not in terms of the difference.

For any $\epsilon > 0$,

$$\lim_{n\to\infty} n^\epsilon = \infty;$$

therefore, we see that

$$\lim_{n\to\infty} \frac{1}{\epsilon \cdot n^\epsilon} = 0,$$

and thus we see that $\log(n)$ grows more slowly than any n^ϵ. The same is true for $\log(n) \cdot \log(n)$ and for $\log(\log(n))$. We can continue inductively to find that any product of logarithms or iterated logarithms grow more slowly than n^ϵ.

We can use the same notation to talk about functions that are substantially smaller than quadratic: $n^{1.5} \in O(n^{2-\epsilon})$. Note that $\frac{n^2}{\log(n)} \notin O(n^{2-\epsilon})$, because dividing by the logarithm is not as significant as subtracting the power by ϵ.

2.7 Practice and discussion questions

1. Find the tightest $O(\cdot)$ bounds for each of the following runtimes. Simplify each of your results as much as possible.

   $2n^2 + n\log(n)$

   $6n\log(n^2) + 5n^2\log(n)$

   $3n + 1.5n^3\frac{\log(n)}{\log(\log(n))}$

   $7.73n^3\frac{\log(n)}{\log(\log(n))} + 0.0001\frac{n!}{2^n}$

   $1 + 2 + 3 + \cdots + n - 1 + n$

   $n\log(n) + \frac{n}{2}\log(n) + \frac{n}{4}\log(n) + \cdots 2\log(n) + \log(n)$

2. What is the exact runtime (*i.e.*, not the big-oh) for each person in a room of n people to shake hands if each handshake takes one step and people cannot shake hands in parallel?

3. What is the tightest $O(\cdot)$ bound of the answer from the previous question?

4. Use the definition of $O(\cdot)$ to find an N and C proving that the $O(\cdot)$ in the previous question is correct.

5. Which of the following functions are $\in O(n^2 \log(n))$?

   $18n^2 \log(\log(n))$

   $\frac{7n^2}{\log(\log(n))}$

   $\frac{3n^{2.5}}{\log(n)}$

   $\frac{1000n^2 \log(n)}{\log(\log(n))}$

   $n(\log(n))^{10}$

   $n\sqrt{n}(\log(n))^{10000}$

6. Which of the following functions are $\in o(n^2)$?

   $18n^2 \log(\log(n))$

   $\frac{7n^2}{\log(\log(n))}$

   $\frac{3n^{2.5}}{\log(n)}$

   $\frac{1000n^2 \log(n)}{\log(\log(n))}$

   $n(\log(n))^{10}$

$n\sqrt{n}(\log(n))^{10000}$

7. Which of the following functions are $\in \Omega(n^2 \log(n))$?

$18n^2 \log(\log(n))$

$\frac{7n^2}{\log(\log(n))}$

$\frac{3n^{2.5}}{\log(n)}$

$\frac{1000n^2 \log(n)}{\log(\log(n))}$

$n(\log(n))^{10}$

$n\sqrt{n}(\log(n))^{10000}$

8. Which of the following functions are $\in \omega(n^2 \log(n))$?

$18n^2 \log(\log(n))$

$\frac{7n^2}{\log(\log(n))}$

$\frac{3n^{2.5}}{\log(n)}$

$\frac{1000n^2 \log(n)}{\log(\log(n))}$

$n(\log(n))^{10}$

$n\sqrt{n}(\log(n))^{10000}$

9. An algorithm is known to run in at most $10n^2 \log(n^8)$ steps. The same algorithm is known to run in at least $\frac{n^2}{2} \log\left(n\sqrt{n}\right)$ steps. Find the tightest $O(\cdot)$ and $\Omega(\cdot)$ for this algorithm. Can we find a $\Theta(\cdot)$ bound?

10. Is there a little-θ set? Why or why not?

11. Give an example of a function $f(n)$ where $f(n) \in O(n^{3+\epsilon})$ but where $f(n) \notin O(n^3)$.

12. Which is a more strict statement: $f(n) \in o(n^k)$ or $f(n) \in O(n^{k-\epsilon})$?

13. Does there exist any function $f(n)$ where $f(n) \in o(n^2)$ but where $f(n) \notin O(n^{2-\epsilon})$? If not, explain why not. If so, give an example of such an $f(n)$.

2.8 Answer key

1. Find the tightest $O(\cdot)$ bounds for each of the following runtimes. Simplify each of your results as much as possible.

   $2n^2 + n\log(n) \in O(n^2)$

   $6n\log(n^2) + 5n^2\log(n) \in O(n^2\log(n))$

   $3n + 1.5n^3\frac{\log(n)}{\log(\log(n))} \in O\left(n^3\frac{\log(n)}{\log(\log(n))}\right)$

   $7.73n^3\frac{\log(n)}{\log(\log(n))} + 0.0001\frac{n!}{2^n} \in O(\frac{n!}{2^n})$. Note that $\frac{n!}{2^n}$ becomes much larger than $n^{3+\epsilon}$ (which itself is even larger than $n^3\frac{\log(n)}{\log(\log(n))}$) as n grows, because

   $$\begin{aligned}\frac{n!}{2^n} &= \frac{\prod_{i=1}^{n} i}{\prod_{i=1}^{n} 2} \\ &= \prod_{i=1}^{n} \frac{i}{2} \\ &= \frac{1}{2} \cdot 1 \cdot \frac{3}{2} \cdot 2 \cdots \frac{n-3}{2} \cdot \frac{n-2}{2} \cdot \frac{n-1}{2} \cdot \frac{n}{2}.\end{aligned}$$

   This polynomial has only one term < 1 (the $\frac{1}{2}$ term) and has a high power; computing the product above from right to left, we

clearly have an n^4 term as $n \to \infty$. The relationship between $n!$ and 2^n will be discussed in greater detail in Chapter 5.

$1 + 2 + 3 + \cdots + n - 1 + n \in O(n^2)$

$n \log(n) + \frac{n}{2} \log(n) + \frac{n}{4} \log(n) + \cdots 2 \log(n) + \log(n) \in O(n \log(n))$

2. The number of ways for people in a room of n people to shake hands is $\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n \cdot (n-1)}{2}$.

3. $\in O(n^2)$.

4. $N = 1$, $C = 1$.

5. The following functions are $\in O(n^2 \log(n))$:

$18n^2 \log(\log(n))$

$\frac{7n^2}{\log(\log(n))}$

$\frac{1000n^2 \log(n)}{\log(\log(n))}$

$n(\log(n))^{10}$

$n\sqrt{n}(\log(n))^{10000}$

6. The following functions are $\in o(n^2)$:

$\frac{7n^2}{\log(\log(n))}$

$n(\log(n))^{10}$

$n\sqrt{n}(\log(n))^{10000}$

7. The following function is $\in \Omega(n^2 \log(n))$:

$\frac{3n^{2.5}}{\log(n)}$

8. The following function is $\in \omega(n^2 \log(n))$:

$\frac{3n^{2.5}}{\log(n)}$

9. $f(n) \in O(n^2 \log(n)) \wedge f(n) \in \Omega(n^2 \log(n)) \to f(n) \in \Theta(n^2 \log(n))$.

10. There is no little-θ set. $O(\cdot)$ indicates all functions bounded above by a ceiling. $\Omega(\cdot)$ indicates all functions bounded below by a floor. $o(\cdot)$ and $\omega(\cdot)$ indicate that the ceiling and floor become infinitely high and low as $n \to \infty$ (using a ratio). There can be no little-θ, because it is impossible for the same function to serve as both an infinitely loose ceiling and an infinitely loose floor.

11. *E.g.*, $f(n) = n^3 \log(n)$.

12. $f(n) \in O(n^{k-\epsilon})$ is stronger; decreasing the exponent must make the ceiling grow infinitely large (meaning we're in $o(\cdot)$); however, the converse is not always true.

13. *E.g.*, $f(n) = \frac{n^2}{\log(n)}$.

Chapter 3

Amortized Analysis

In some cases, the worst-case of an individual operation may be quite expensive, but it can be proven that this worst-case cannot occur frequently. For example, consider a stack data structure that supports operations: `push`, `pop`, and `pop_all`, where `push` and `pop` behave in the standard manner for a stack and where `pop_all` iteratively pops every value off of the stack using the `pop` operation (Listing 3.1). The worst-case runtimes of `push` and `pop` will be constant, while `pop_all` will be linear in the current size of the stack (Table 3.1).

If someone asked us what was the worst-case runtime of any operation on our stack, the answer would be $O(n)$; however, if they asked us what would be the worst-case runtime of *several* operations on an initially empty stack, the answer is more nuanced: although one individual operation performed may prove to be expensive, we can prove that this cannot happen often. This is the basis of "amortized analysis".

Operation	Worst-case runtime
`push`	1
`pop`	1
`pop_all`	n

Table 3.1: Worst-case runtimes of operations on a stack currently holding n items.

Listing 3.1: Simple stack.

```
class Stack:
    def __init__(self):
        self._data = []

    # push runs in O(1) under the assumptions listed:
    def push(self, item):
        # assume O(1) append (this can be guaranteed using a linked
        # list implementation):
        self._data.append(item)

    # pop runs in O(1) under the assumptions listed:
    def pop(self):
        item = self._data[-1]
        # assume O(1) list slicing (this can be guaranteed using a
        # linked list implementation):
        self._data = self._data[:-1]
        return item

    # push runs in worst-case O(n) where n=len(self._data);
    # however, the worst-case amortized runtime is in O(1)
    def pop_all(self):
        while self.size() > 0:
            self.pop()

    def size(self):
        return len(self._data)
```

Operation	Amortized worst-case runtime
`push`	$\widetilde{O}(1)$
`pop`	$\widetilde{O}(1)$
`pop_all`	$\widetilde{O}(1)$

Table 3.2: Worst-case amortized runtimes of operations on a stack.

3.1 Accounting method

When operating on an initially empty stack, each `pop` step performed by `pop_all` *must* have been proceeded by a completed `push` operation; after all, if we're popping values from the stack, they must have been pushed at some point prior. Thus, we can use the "accounting method" to pay for that work in advance. Thus, we will adjust that a single `push` operation takes 1 step plus our advance payment of the 1 step of work necessary if we ever call `pop_all`. Because `pop_all` has been paid for in advance, it is essentially free. Likewise, `pop` operations are free for the same reason (because we have pre-allocated the cost in advance by paying during the `push` operations). Thus we can see that the cost of `push` is $2 \in O(1)$, and the cost of `pop` and `pop_all` are both $0 \in O(1)$. This leads to inexpensive "amortized" costs (Table 3.2).

Low amortized costs do not guarantee anything about an individual operation; rather, they guarantee bounds on the average runtime in any long sequence of operations.

3.2 Potential method

The "potential method" is an alternative, more complex approach to the accounting method for deriving amortized bounds. In the potential method, a "potential" function Φ is used. This potential function computes a numeric value from the current state of our stack.

At iteration i, let the state of our stack be noted S_i. $\Phi(S_i)$ is a numeric value computing the potential of S_i. Think of the potential function as stored-up "work debt". This is similar to the accounting method; however, the potential method can behave in a more complex manner than in the accounting method. Let the cost of the operation performed at iteration i be denoted c_i. If we construct our potential function so that the potential of our initial data structure is less than or equal to the potential after running

n iterations, (*i.e.*, $\Phi(S_n) \geq \Phi(S_0)$), then the total runtime of executing n sequential operations, $\sum_{i=1}^{n} c_i$, can be bounded above[1]:

$$\begin{aligned}\sum_{i=1}^{n} c_i + \Phi(S_i) - \Phi(S_{i-1}) &= \left(\sum_{i=1}^{n} c_i\right) + \Phi(S_n) - \Phi(S_0) \\ &\geq \sum_{i=1}^{n} c_i.\end{aligned}$$

Thus, we can use $\widehat{c_i} = c_i + \Phi(S_i) - \Phi(S_{i-1})$ as a surrogate cost and guarantee that we will still achieve an upper bound on the total cost.

We will need to choose our potential function strategically. In the case of our stack, we can choose our potential function $\Phi(S_i) = S_i.\text{size}()$, which holds our necessary condition that $\Phi(S_n) \geq \Phi(S_0)$ (because we start with an empty stack).

The upper bound of the amortized cost of a `push` operation will be

$$\begin{aligned}\widehat{c_i} = c_i + \Phi(S_i) - \Phi(S_{i-1}) &= 1 + S_i.\text{size}() - S_{i-1}.\text{size}() \\ &= 1 + 1 = 2 \in O(1),\end{aligned}$$

because a `push` operation increases the stack size by 1. Likewise, an upper bound on the amortized cost of a `pop` operation will be

$$\begin{aligned}\widehat{c_i} = c_i + \Phi(S_i) - \Phi(S_{i-1}) &= 1 + S_i.\text{size}() - S_{i-1}.\text{size}() \\ &= 1 + S_{i-1}.\text{size}() - 1 - S_{i-1}.\text{size}() \\ &= 1 - 1 = 0 \in O(1),\end{aligned}$$

because a `pop` operation will decrease the stack size by 1. An upper bound on the amortized cost of a `pop_all` operation will be

$$\begin{aligned}\widehat{c_i} = c_i + \Phi(S_i) - \Phi(S_{i-1}) &= S_{i-1}.\text{size}() + S_i.\text{size}() - S_{i-1}.\text{size}() \\ &= S_i.\text{size}() = 0 \in O(1),\end{aligned}$$

because the size after running `multi_pop` will be $S_i = 0$. Thus we verify our result with the accounting method and demonstrate that every operation is $\in \widetilde{O}(1)$.

[1]This is called a "telescoping sum" because the terms collapse down as sequential terms cancel, just like collapsing a telescope.

The potential method is more flexible than the accounting method, because the "work" stored up can be modified freely (using any symbolic formula) at runtime; in contrast, the simpler accounting method accumulates the stored-up work in a static manner in advance.

3.3 Vectors: resizable arrays

Consider the vector, a data structure that behaves like a contiguous array, but which allows us to dynamically append new elements. Every time an array is resized, a new array must be allocated, the existing data must be copied into the new array, and the old array must be freed[2]. Thus if you implement an `append` function by growing by only 1 item each time `append` is called, then the cost of n successive `append` operations will be

$$1 + 2 + 3 + \cdots + n - 1 + n \in \Theta(n^2).$$

This would be inferior to simply using a linked list, which would support $O(1)$ `append` operations.

However, if each time an `append` operation is called, we grow the vector by more than we need each, then one expensive resize operation will guarantee that the following `append` operations will be inexpensive. If we grow the vector exponentially, then we can improve the amortized cost of n `append` operations[3]

If we grow by 1 during each `append` operation, the runtime of performing n operations is $\Theta(n^2)$, and thus the amortized cost of each operation is $\in \widetilde{O}(\frac{n^2}{n}) = \widetilde{O}(n)$ per operation.

On the other hand, if we grow by doubling, we can see that a resize operation that resizes from capacity s to $2s$ will cost $O(s)$ steps, and that this resize operation will guarantee that the subsequent $s - 1$ `append` operations each cost $O(1)$. Consider the cost of all resize operations: to insert n items, the final capacity will be $\leq 2n$ (because we have an invariant that the size is never less than half the capacity). Thus, the cost of all resize operations will

[2]For simplicity, ignore the existence of the `realloc` function in `C` here.

[3]For a complete derivation of why exponential growth is necessary and practical performance considerations, see Chapter 6 of "Code Optimization in `C++11`" (Serang 2018).

be

$$\begin{aligned} &\leq 2n + n + \frac{n}{2} + \frac{n}{4} + \cdots + 4 + 2 + 1 \\ &< 2n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right) \\ &= 4n \\ &\in O(n). \end{aligned}$$

The total cost will be the cost of actually inserting the n items plus the cost of all resize operations, which will be $O(n) + O(n) = O(n)$. Since all n append operations can be run in $O(n)$ total, then the amortized cost per each append will be $\widetilde{O}(\frac{n}{n}) = \widetilde{O}(1)$.

3.3.1 Demonstration of our $\widetilde{O}(1)$ vector doubling scheme

In order to grow our vector by doubling, at every append operation we will check whether the vector is full (*i.e.*, whether the size that we currently use in our vector has reached the full capacity we have allocated thus far). If it is full, we will resize to double the capacity. On the other hand, if our vector is not yet full, we will simply insert the new element. In Listing 3.2, the class `GrowByDoublingVector` implements this doubling strategy, while `GrowBy1Vector` uses a naive approach, growing by 1 each time (which will take $O(n^2)$ time). The output of the program shows that appending 10000 elements by growing by 1 takes 2.144 seconds, while growing by doubling takes 0.005739s.

Listing 3.2: Two vector implementations. `GrowBy1Vector` grows by 1 during each `append` operation, while `GrowByDoublingVector` grows by doubling its current capacity.

```
from time import time

class Vector:
    def __init__(self):
        self._size=0

    def size(self):
        return self._size
```

```
class GrowBy1Vector(Vector):
    def __init__(self):
        Vector.__init__(self)
        self._data = []

    # n successive append operations will cost O(n^2) total
    def append(self, item):
        new_capacity = self.size()+1
        new_data = [None]*new_capacity

        # copy in old data:
        for i in xrange(self._size):
            new_data[i] = self._data[i]

        self._data = new_data

        self._data[self.size()] = item
        self._size += 1

    def get_data(self):
        return self._data

class GrowByDoublingVector(Vector):
    def __init__(self):
        Vector.__init__(self)
        # start with a capacity of 1 element:
        self._data = [None]

    # n successive append operations will cost O(n) total
    def append(self, item):
        if self.size() == self.capacity():
            # make sure we grow initially if the capacity is 0 (which
            # would double to 0) by taking the max with 1:
            new_capacity = 2*self.capacity()
            new_data = [None]*new_capacity

            # copy in old data:
            for i in xrange(self._size):
                new_data[i] = self._data[i]

            self._data = new_data

        self._data[self.size()] = item
        self._size += 1
```

```
    def capacity(self):
        return len(self._data)

    def get_data(self):
        return self._data[:v.size()]

N=10000

t1=time()
v = GrowBy1Vector()
for i in xrange(N):
    v.append(i)
#print v.get_data()
t2=time()
print 'Growing by 1 took', t2-t1, 'seconds'

t1=time()
v = GrowByDoublingVector()
for i in xrange(N):
    v.append(i)
#print v.get_data()
t2=time()
print 'Growing by doubling took', t2-t1, 'seconds'
```

Chapter 4

Selection Sort and Merge Sort

4.1 Selection sort

Selection sort is one of the simplest sorting algorithms. Its premise is simple: First, find the minimum element in the list (by visiting all indices $0, 1, 2, 3, \ldots$). This will be `result[0]`. Second, find the second smallest value in the list (this is equivalent to finding the smallest value from indices $1, 2, 3, \ldots$ (because index 0 now contains the smallest value). Move this to `result[1]`. Third, find the smallest value from indices $2, 3, \ldots$. Move this to `result[2]`.

Continuing in this manner, we can see that this algorithm will sort the list: the definition of sorted order is that the smallest value will be found in `result[0]`, the second smallest value will be found in `result[1]`, *etc.* This is shown in Listing 4.1.

4.1.1 Derivation of runtime #1

The runtime of this algorithm can be found by summing the cost of each step: the cost to find the minimum element in all n values, the cost to find the minimum element in $n-1$ values, the cost to find the minimum element in $n-2$ values, *etc.* This will cost $n-1 \ + \ n-2 \ + \ \cdots \ + \ 3 \ + \ 2 \ + \ 1$ steps, each of which cost $\Theta(1)$ (because they are `if` statements, primitive copy operations, *etc.*). If we pair the terms from both ends, we see that this

equals

$$\begin{aligned} & n-1+1 \ + \ n-2+2 \ + \ n-3+3 \ + \cdots \\ = \ & \underbrace{n \ + \ n \ + \ \cdots \ + \ n}_{\frac{n-1}{2} \text{ terms}} \\ = \ & \frac{n \cdot (n-1)}{2} \\ \in \ & \Theta(n^2). \end{aligned}$$

Thus we see that selection sort consists of $\Theta(n^2)$ operations, each of which cost $\Theta(1)$, and so selection sort is $\in \Theta(n^2)$.

4.1.2 Derivation of runtime #2

We can also see that selection sort consists of two nested loops, one looping i in $0, 1, \ldots n-1$ and the other looping j in $i+1, i+2, \ldots, n-1$. Together, the number of (i, j) pairs visted will be $|\{(i, j) : 0 \leq i < j < n\}|$. Because we know that $i < j$, for any set $\{i, j\}$ where $i \neq j$, we can figure out which index is i and which index is j (note that sets are unordered, so $\{i, j\} = \{j, i\}$). For example, if $\{i, j\} = \{3, 2\}$, then $i = 2$ and $j = 3$ is the only solution that would preserve $i < j$; Therefore,

$$\begin{aligned} & |\{(i, j) : 0 \leq i < j < n\}| \\ = \ & |\{\{i, j\} : i \neq j \ \wedge \ i, j \in \{0, 1, 2, \ldots, n-1\}\}| \\ = \ & \binom{n}{2}. \end{aligned}$$

$\binom{n}{2} = \frac{n \cdot (n-1)}{2} \in \Theta(n^2)$. Thus, these nested `for` loops will combine to perform $\Theta(n^2)$ iterations, each of which cost $\Theta(1)$ (because they only use `if` statements, primitive copy operations, *etc.*). This validates our result above that selection sort $\in \Theta(n^2)$.

Listing 4.1: Selection sort.

```
# costs O(1) per {i,j} pair where i and j are in {0, 1, ... n-1} and
# j>i. this will cost n choose 2, which is \in \Theta(n^2).
def selection_sort(arr):
    n=len(arr)
    # make a local copy to modify and sort:
```

```
    result = list(arr)

    # compute result[i]
    for i in xrange(n):
        # find the minimum element in all remaining
        min_index=i

        for j in xrange(i+1,n):
            if result[j] < result[min_index]:
                min_index=j

        # swap(result[min_index], result[j])
        temp=result[min_index]
        result[min_index]=result[i]
        result[i]=temp

        # result[i] now contains the minimum value in the remaining
        # array
        print result

    return result

print selection_sort([10,1,9,5,7,8,2,4])
```

4.2 Merge sort

Merge sort is a classic divide-and-conquer algorithm. It works by recursively sorting each half of the list and then merging together the sorted halves.

In each recursion, merge sort of size n calls two merge sorts of size $\frac{n}{2}$ and then performs merging in $\Theta(n)$. Thus we have the recurrence $r(n) = 2r(\frac{n}{2}) + \Theta(n)$. Later, we will see how to solve this recurrence using the Master Theorem, but for now, we can solve it using calculus.

The overhead[1] of each recursive call will be in $\Theta(1)$. Therefore, let us only consider the cost of merging (which eclipses the overhead of invoking the 2 recursive calls). If we draw a recursive call tree, we observe a cost of $\Theta(n)$ at the root node, and a split into two recursive call nodes, each of which will cost $\Theta(\frac{n}{2})$. These will split in a similar fashion.

From this we can see that the cost of each layer in the tree will be in $\Theta(n)$. For example, the recursive calls after the root will cost $\Theta(\frac{n}{2}) + \Theta(\frac{n}{2}) = \Theta(n)$.

[1] *E.g.*, of copying parameters to the stack and copying results off of the stack.

From this we see that the total runtime will be bounded by summing over the cost of each layer ℓ:

$$\begin{aligned}\sum_{\ell=0}^{L-1}\Theta(n) &= \Theta(nL)\\ &= \Theta(n\log(n)),\end{aligned}$$

because L, the number of layers in the tree, will be $\log_2(n)$. The runtime of merge sort is $\in \Theta(n\log(n))$.

Listing 4.2: Merge sort.

```
# costs r(n) = 2r(n/2) + \Theta(n) \in \Theta(n log(n))
def merge_sort(arr):
    n=len(arr)
    # any list of length 1 is already sorted:
    if n <= 1:
        return arr

    # make copies of the first and second half of the list:
    first_half = list(arr[:n/2])
    second_half = list(arr[n/2:])

    first_half = merge_sort(first_half)
    second_half = merge_sort(second_half)

    # merge
    result = [None]*n
    i_first=0
    i_second=0
    i_result=0
    while i_first < len(first_half) and i_second < len(second_half):
        if first_half[i_first] < second_half[i_second]:
            result[i_result] = first_half[i_first]
            i_first += 1
            i_result += 1
        elif first_half[i_first] > second_half[i_second]:
            result[i_result] = second_half[i_second]
            i_second += 1
            i_result += 1
        else:
            # both values are equal:
            result[i_result] = first_half[i_first]
            result[i_result+1] = second_half[i_second]
```

```
            i_first += 1
            i_second += 1
            i_result += 2

    # insert any remaining values:
    while i_first < len(first_half):
        result[i_result] = first_half[i_first]
        i_result += 1
        i_first += 1

    while i_second < len(second_half):
        result[i_result] = second_half[i_second]
        i_result += 1
        i_second += 1

    return result

print merge_sort([10,1,9,5,7,8,2,4])
```

Chapter 5

Quicksort

Quicksort is famous because of its ability to sort in-place. *I.e.*, it directly modifies the contents of the list and uses $O(1)$ temporary storage. This is not only useful for space requirements (recall that our merge sort used $O(n \log(n))$ space), but also for practical efficiency: the allocations performed by merge sort prevent compiler optimizations and can also result in poor cache performance[1]. Quicksort is often a favored sorting algorithm because it can be quite fast in practice.

Quicksort is a sort of counterpoint to merge sort: Merge sort sorts the two halves of the array and then merges these sorted halves. In contrast, quicksort first "pivots" by partitioning the array so that all elements less than some "pivot element" are moved to the left part of the array and all elements greater than the pivot element are moved to the right part of the array. Then, the right and left parts of the array is recursively sorted using quicksort and the pivot element is placed between them (Listing 5.2).

Runtime analysis of quicksort can be more tricky than it looks. For example, we can see that if we choose the pivot elements in ascending order, then during pivoting, the left part of the array will always be empty while the right part of the array will contain all remaining $n - 1$ elements (*i.e.*, it excludes the pivot element). The cost of pivoting at each layer in the call tree will therefore be $\Theta(n), \Theta(n-1), \Theta(n-2), \ldots, \Theta(1)$, and the total cost will be $\in \Theta(n + n - 1 + n - 2 + \cdots + 1) = \Theta(n^2)$. This poor result is because the recursions are not balanced; our divide and conquer does not divide effectively, and therefore, it hardly conquers.

[1] See Chapter 3 of "Code Optimization in `C++11`" (Serang 2018)

5.1 Worst-case runtime using median pivot element

One approach to improving the quicksort runtime would be choosing the median as the pivot element. The median of a list of length n is guaranteed to have $\frac{n-1}{2}$ values $\leq$ to it and to have $\frac{n-1}{2}$ values $\geq$ to it. Thus the runtime of using the median pivot element would be given by the recurrence

$$\begin{aligned} r(n) &= 2r\left(\frac{n-1}{2}\right) + \Theta(n), \\ &< 2r\left(\frac{n}{2}\right) + \Theta(n), \end{aligned}$$

where the $\Theta(n)$ cost comes from the pivoting step. This matches the runtime recurrence we derived for merge sort, and we can see that using quicksort with the median as the pivot will cost $\Theta(n \log(n))$.

There is a large problem with this strategy: we have assumed that the cost of computing the median is trivial; however, if asked to compute a median, most novices would do so by sorting and then choosing the element in the middle index (or one of the two middle indices if n should happen to be even). It isn't a good sign if we're using the median to help us sort and then we use sorting to help compute the median. There is an $O(n)$ divide-and-conquer algorithm for computing the median, but that algorithm is far more complex then quicksort. Regardless, even an available linear-time median algorithm would add significant practical overhead and deprive quicksort of its magic.

5.2 Expected runtime using random pivot element

We know that quicksort behaves poorly for some particular pivoting scheme and we also know that quicksort performs well in practice. Together, these suggest that a randomized algorithm could be a good strategy. Now, obviously, the worst-case performance when choosing a random pivot element would still be $\Omega(n^2)$, because our random pivot elements may correspond to visiting pivots in ascending order (or descending order, which would also yield poor results).

Let us consider the expected runtime[2]. We denote the operation where elements i and j are compared using the random variable $C_{i,j}$:

$$C_{i,j} = \begin{cases} 1 & i \text{ and } j \text{ are compared} \\ 0 & \text{else} \end{cases}.$$

The total number of comparisons will therefore be the sum of comparisons on all unique pairs:

$$\sum_{\{i,j\}:i\neq j} C_{i,j}.$$

Excluding the negligable cost of the recursion overhead, the cost of quicksort will be the cost of all comparisons plus the cost of all swaps, and since each comparison produces at most one swap, use the accounting method to simply say that the runtime will be bounded above by this total number of comparisons.

The expected value of the number of comparisons will be the sum of the expected values of the individual comparisons:

$$\mathbb{E}\left[\sum_{\{i,j\}:i\neq j} C_{i,j}\right] = \sum_{\{i,j\}:i\neq j} \mathbb{E}\left[C_{i,j}\right].$$

The expected value of each comparison will be governed by $p_{i,j}$, the probability of whether elements i and j are ever compared:

$$\mathbb{E}\left[C_{i,j}\right] = p_{i,j} \cdot 1 + (1 - p_{i,j}) \cdot 0 = p_{i,j}.$$

Note that if two values are ever pivoted to opposite sides of the array, they will never be in the same array during further recursive sorts, and thus they can never be compared. This means that if any element x with a value between elements i and j is ever chosen as the pivot element before i or j are chosen as the pivot element, then i and j will never be compared. Since the pivot elements are chosen randomly, then from the perspective of elements i and j, then the probability they will be compared will be $\frac{2}{|r_j - r_i| + 1}$, where r_i gives the index of element i in the sorted list.

[2] *I.e.*, the average runtime if our quicksort implementation were called on any array filled with unique values.

If our list contains unique elements[3], then we can also sum over the comparisons performed by summing over the ranks rather than the elements (in both cases, we visit each pair exactly once):

$$\begin{aligned}
\text{total runtime} &\propto \sum_{\{i,j\}:i\neq j} \mathbb{E}\left[C_{i,j}\right] \\
&= \sum_{\{i,j\}:i\neq j} p_{i,j} \\
&= \sum_{\{r_i,r_j\}:r_i\neq r_j} \frac{2}{|r_j - r_i| + 1} \\
&= \sum_{r_i} \sum_{r_j > r_i} \frac{2}{r_j - r_i + 1}.
\end{aligned}$$

We transform into a more accessible form by letting $k = r_j - r_i + 1$:

$$\begin{aligned}
&= \sum_{r_i} \sum_{k=r_j-r_i+1:r_j>r_i\wedge r_j\leq n} \frac{2}{k} \\
&\leq \sum_{r_i} \sum_{k=2}^{n} \frac{2}{k} \\
&= \sum_{k=2}^{n} \sum_{r_i=1}^{n} \frac{2}{k} \\
&= 2n \sum_{k=2}^{n} \frac{1}{k}.
\end{aligned}$$

Here we use the fact that $\sum_{k=1}^{n} \frac{1}{k}$ is a "harmonic sum", which can be seen as an approximation of

$$\int_1^n \frac{1}{x} \partial x.$$

Specifically, if we plot the continuous function $\frac{1}{x}, x \geq 1$ and compare it to the discretized $\frac{1}{\lfloor x \rfloor}$, we see that $\frac{1}{\lfloor x \rfloor}$ will never underestimate $\frac{1}{x}$ because $x \geq \lfloor x \rfloor$ and thus $\frac{1}{x} \leq \frac{1}{\lfloor x \rfloor}$ (Figure 5.1). If we shift the discretized function left by 1

[3]Quicksort works just as well when we have duplicate elements, but this assumption simplifies the proof.

to yield $\frac{1}{\lfloor x \rfloor + 1}$, we see that it never overestimates $\frac{1}{x}$ (because $x < \lfloor x \rfloor + 1$ and thus $x > \frac{1}{\lfloor x \rfloor + 1}$). It follows that the area

$$\begin{aligned} \sum_{k=1}^{n} \frac{1}{k+1} &= \int_1^n \frac{1}{\lfloor x \rfloor + 1} \partial x \\ &< \int_1^n \frac{1}{x} \partial x. \\ &= \log_e(n). \end{aligned}$$

We can rewrite

$$\begin{aligned} \sum_{k=2}^{n+1} \frac{1}{k} &= \sum_{k=1}^{n} \frac{1}{k+1} \\ &< \log_e(n). \end{aligned}$$

Our harmonic sum, $\sum_{k=2}^{n} \frac{1}{k}$, is bounded above by a sum with an additional nonnegative term, $\sum_{k=2}^{n+1} \frac{1}{k} < \log_e(n)$. And hence, our expected quicksort runtime is bounded above by $2n \log_e(n) \in O(n \log(n))$.

Listing 5.1: Quicksort.

```
import numpy as np

def swap_indices(arr, i, j):
    temp = arr[i]
    arr[i] = arr[j]
    arr[j] = temp

def quicksort(arr, start_ind, end_ind):
    n=end_ind - start_ind + 1
    # any list of length 1 is already sorted:
    if n <= 1:
        return

    # choose a random pivot element in {start_ind, ..., end_ind}
    pivot_index = np.random.randint(start_ind, end_ind+1)
    pivot = arr[pivot_index]

    # count values < pivot:
    vals_lt_pivot=0
```

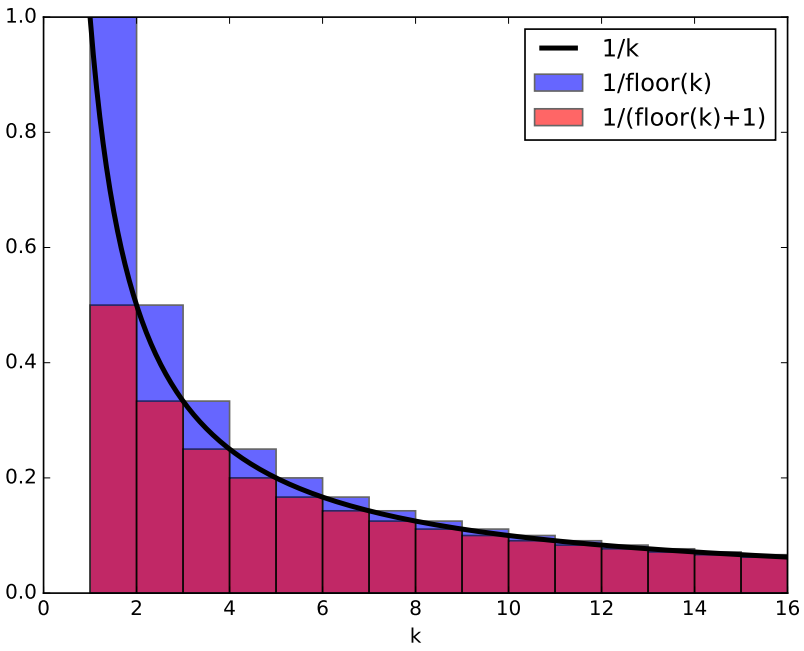

Figure 5.1: Illustration of a harmonic sum. $\frac{1}{\lfloor x \rfloor}$ is an upper bound of $\frac{1}{x}$ while $\frac{1}{\lfloor x \rfloor + 1}$ is a lower bound of $\frac{1}{x}$.

```
    for i in xrange(start_ind, end_ind+1):
        if arr[i] < pivot:
            vals_lt_pivot += 1

    # place pivot at arr[vals_lt_pivot], since vals_lt_pivot will need to
        come before it
    swap_indices(arr, pivot_index, start_ind+vals_lt_pivot)
    # the pivot index has been moved to index vals_lt_pivot
    pivot_index = start_ind+vals_lt_pivot

    # move all values < pivot to indices < pivot_index:
    vals_lt_pivot=0
    for i in xrange(start_ind, end_ind+1):
        if arr[i] < pivot:
            swap_indices(arr, i, start_ind+vals_lt_pivot)
            vals_lt_pivot += 1

    # pivoting is complete. recurse:
    quicksort(arr, start_ind, start_ind+vals_lt_pivot)
    quicksort(arr, start_ind+vals_lt_pivot+1, end_ind)

arr = [10,1,9,5,7,8,2,4]
quicksort(arr, 0, len(arr)-1)
print arr
```

5.3 Practice and discussion questions

1. Let $x = [9, 1, 7, 8, 2, 5, 4, 3]$ and $i = 2$ so that $x[i]$ is 7. Using `Python`, use the `list.index` function to compute r_i, the index occupied by $x[i]$ in the sorted list `x_sort = sorted(x)`.

2. Using the formula derived in the notes, what is the probability that we will compare indices $i = 2$ and $j = 4$, *i.e.*, compute $Pr(C_{2,4})$. Implement this using a function `probability_of_comparison(x,x_sort,i,j)`.

3. What is the expected number of operations that will be used to compare the value at $i = 2$ to the value at $j = 4$?

4. In your `Python` program, create a function to compute the expected number of comparisons between all indices i and all $j > i$ using quick-

sort:

$$\mathbb{E}\left[\sum_{i,j} C_{i,j}\right] = \sum_{i,j:j>i} \Pr(C_{i,j}).$$

Each time you compute $\Pr(C_{i,j})$, simply call your `probability_of_comparison(x,x_sort,i,j)` function.

5. Repeat this using lists of unique elements of size $n \in \{16, 32, 64, \ldots 1024\}$ using `matplotlib` (or `pylab`), plot the expected number of comparisons (y-axis) against n (x-axis) using a log-scale for both axes. Add a series containing n, a series containing $n \log_2(n)$, and a series containing n^2. How do your series of expected runtimes compare?

5.4 Answer key

Listing 5.2: Quicksort.

```
import numpy
import pylab

# x should have unique elements for this line of analysis:
x = [9,1,7,8,2,5,4,3]
i=2
j=4

x_sort = sorted(x)
print 'original', x
print 'sorted ', x_sort

r_i = x_sort.index(x[i])
print x[i], 'lives at index', r_i, 'in x_sort'
r_j = x_sort.index(x[j])
print x[j], 'lives at index', r_j, 'in x_sort'
print

def empirical_probability_of_comparison(x, x_sort, i, j):
  smaller_val = min(x[i], x[j])
  larger_val = max(x[i], x[j])

  number_pivots_between = 0
  number_pivots_comparing_values_at_i_and_j = 0
```

```
  for val in x_sort:
    if val >= smaller_val and val <= larger_val:
      number_pivots_between += 1
    if val == smaller_val or val == larger_val:
      number_pivots_comparing_values_at_i_and_j += 1
    # pivot selections outside of {i, i+1, ... , j} do not affect whether
        we compare

  return float(number_pivots_comparing_values_at_i_and_j) /
      number_pivots_between

def algebraic_probability_of_comparison(x, x_sort, i, j):
  r_i = x_sort.index(x[i])
  r_j = x_sort.index(x[j])

  return 2.0 / (numpy.fabs(r_j-r_i) + 1)

# These should match:

# Estimate C_{i,j} empirically:
print 'Empirical estimate of Pr(C_{i,j}=1):',
    empirical_probability_of_comparison(x,x_sort,i,j)
# Estimate C_{i,j} using formula in the notes:
print 'Algebraic estimate of Pr(C_{i,j}=1):',
    algebraic_probability_of_comparison(x,x_sort,i,j)
print

def total_expected_comparisons(x, x_sort):
  result = 0.0
  for i in range(len(x)):
    for j in range(i+1, len(x)):
      result += algebraic_probability_of_comparison(x, x_sort, i, j)
  return result

print 'Total exected number of comparisons:',
    total_expected_comparisons(x, x_sort)

n_series = 2**numpy.arange(6, 11)
comparison_series = []
for n in n_series:
  y = range(n)
  # note: y is already sorted; we sort for readability here:
  expected_comparisons = total_expected_comparisons(y, sorted(y))
  print n, expected_comparisons
  comparison_series.append(expected_comparisons)
```

```
pylab.plot(n_series, n_series, label='Linear', alpha=0.8)
pylab.plot(n_series, comparison_series, label='Expected quicksort
    comparisons', alpha=0.8)
pylab.plot(n_series, n_series*numpy.log2(n_series), label=r'$n \cdot
    log_2(n)$', alpha=0.8)
pylab.plot(n_series, n_series**2, label='Quadratic', alpha=0.8)
pylab.xscale('log')
pylab.yscale('log')
pylab.xlabel('n')
pylab.ylabel('Operations performed')
pylab.legend()
pylab.show()
```

Chapter 6

Runtime bounds on comparison sort

Both merge sort and quicksort only use the $<$ operator between elements to compare them. In this manner, they are both "comparison sorting" algorithms. Non-comparison sorting algorithms exploit additional properties of the numbering elements. *E.g.*, we are able to "peel" the most-significant digit off of an integer to perform postman sort (or the least-significant digit to perform radix sort). But if the only thing we know about our elements is the existence of a $<$ operator, comparison sorts are all that we can do.

Here we consider the worst-case runtime of any comparison sort algorithm, even those that have never been proposed. We would like to know if it is ever possible to do better than $O(n \log(n))$ using a comparison sorting on any array of n elements.

An array of n values can be arranged in $n!$ unique permutations[1]. Now consider a comparison sorting algorithm, which will perform as well as possible against a malicious opponent. That is, this unknown comparison sorting algorithm that we dream of will perform as well as possible against a worst-case input.

If we think abstractly, sorting is the process of mapping any of those $n!$ array permutations into the unique sorted array permutation. In an optimal world, each comparison will contribute 1 additional bit of information[2]. In

[1]For simplicity, let's assume that there are no duplicate elements.

[2]This is not true in general if we choose sub-optimal comparisons to perform. For example, if we know that $a < b$ from one comparison and know that $b < c$ from a second comparison, then the comparison $a < c$ *must* be true, and therefore contributes no

this manner, optimal comparisons will divide the space of $n!$ unsorted arrays in half.

Using this strategy, we can construct a decision tree, which iteratively divides $n!$ in half during each comparison, until we reach the unique sorted ordering. The longest path from the root to any leaf will be the number of comparisons that we need to perform against a malicious opponent. Note that an optimal comparison strategy guarantees that our decision tree will be balanced, which guarantees that any initial ordering chosen by that opponent will not be able to make any path of comparisons taking us from the root to the leaf substantially more expensive than any other. The depth of a balanced binary decision tree with $n!$ leaves will be $\log_2(n!)$. We can expand $\log(n!)$ into the following:

$$\begin{aligned}\log(n!) &= \log(n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1) \\ &= \log(n) + \log(n-1) + \log(n-2) + \cdots + \log(3) + \log(2) + \log(1).\end{aligned}$$

6.1 Lower bound on the number of steps required by an optimal comparison sorting algorithm

Under the assumption of an optimal comparison strategy (even if we are not sure precisely how that would work), the number of comparisons needed to multiplex the $n!$ possible array permutations to the unique sorted ordering will be $log_2(n!)$, with which we can derive a lower bound on the number of

---

additional information.

comparisons that would need to be performed:

$$\begin{aligned}
\log(n!) &= \log(n) + \log(n-1) + \log(n-2) + \cdots + \log(3) + \log(2) + \log(1) \\
&> \log(n) + \log(n-1) + \log(n-2) + \cdots + \log\left(\frac{n}{2}\right) + 0 + \cdots + 0 + 0 + 0 \\
&> \underbrace{\log\left(\frac{n}{2}\right) + \log\left(\frac{n}{2}\right) + \log\left(\frac{n}{2}\right) + \cdots + \log\left(\frac{n}{2}\right)}_{\frac{n}{2} \text{ terms}} + 0 + \cdots + 0 + 0 + 0 \\
&= \frac{n}{2}\log\left(\frac{n}{2}\right) \\
&= \frac{n}{2}\left(\log(n) - \log(2)\right) \\
&\in \Omega(n\log(n)).
\end{aligned}$$

Amazingly, this demonstrates that *no* comparison sort, even one that has never been dreamed up, can have a runtime substantially faster than $n\log(n)$.

6.2 Upper bound on the number of steps required by an optimal comparison sorting algorithm

Using a strategy similar to the one above, we can also derive an upper bound on the number of steps necessary by some hypothetical optimal comparison strategy. This is arguably less satisfying than the pessemistic $\Omega(\cdot)$ bound, because it may be easy to be optimistic under the assumption of some optimal comparison strategy that has not yet been shown. Nonetheless, our upper bound is as follows:

$$\begin{aligned}
\log(n!) &= \log(n) + \log(n-1) + \log(n-2) + \cdots + \log(3) + \log(2) + \log(1) \\
&< \log(n) + \log(n) + \log(n) + \cdots + \log(n) + \log(n) + \log(n) \\
&= n\log(n) \\
&\in O(n\log(n)).
\end{aligned}$$

This verifies what we have seen already: our merge sort and quicksort implementations were comparison sorts, and they were in $O(n\log(n))$. Because we have already seen such algorithms by construction, we no longer need

to worry about our previous constraint that the comparisons be performed optimally. We know that $O(n\log(n))$ is an upper bound on number of steps needed by well-constructed comparison sorts, even in the worst-case (*i.e.*, even against a malicious opponent).

Thus we see that $\log(n!) \in \Theta(n\log(n))$. No comparison sort can be substantially better than $n\log(n)$ steps and no well-constructed comparison sort should be substantially worse than $n\log(n)$ steps. From a runtime perspective, our humble merge sort implementation is within a factor of the optimal possible comparison sort.

6.3 $\frac{n!}{2^n}$ revisited

Consider practice question 1.4 from Chapter 2: there we observed that $7.73n^3\frac{\log(n)}{\log(\log(n))} + 0.0001\frac{n!}{2^n} \in O\left(\frac{n!}{2^n}\right)$. Previously, we did not simplify $O\left(\frac{n!}{2^n}\right)$; however with the derivation in this chapter, we can do so.

If we define the runtime as $r(n) = \frac{n!}{2^n}$, then we can discuss the log runtime

$$\begin{aligned}\log(r(n)) &= \log\left(\frac{n!}{2^n}\right)\\ &= \log(n!) - \log(2^n)\\ &\in \Theta(n\log(n)) - n\\ &= \Theta(n\log(n)).\end{aligned}$$

Thus we see that the log runtime is within a constant of $\log(n!)$, which would be the log runtime of an algorithm with runtime $n!$. Thus, in log-space, $\frac{n!}{2^n}$ and $n!$ are within a constant of one another. By exponentiating, we see that $r(n) \in 2^{\Theta(n\log(n))}$. Because $\Theta(f(n))$ is the set of functions bounded above and below within a constant factor of $f(n)$ when n becomes large, then there exists some constant C such that asymptotically,

$$r(n) < 2^{C\cdot n\log(n)} = \left(2^{n\log(n)}\right)^C$$

and

$$r(n) > 2^{\frac{n\log(n)}{C}} = \left(2^{n\log(n)}\right)^{\frac{1}{C}}.$$

It is not uncommon to see runtimes bounded using notation such as $2^{O(\cdot)}$ or $2^{-\Omega(\cdot)}$.

6.4 Practice and discussion questions

1. Comparison-based sorting can be viewed as sorting n items where our only interaction between the items is with a scale that can hold two items, a and b, and measure whether a is heavier or whether b is heavier; however, thus far we've only discussed the case where the array that we're sorting contains only unique values. Consider how this would work if the scale could report that a is heavier, that b is heavier, or that a and b are of equal weight. What would be a lower bound (*i.e.*, $\Omega(\cdot)$) for the best-case runtime (*i.e.*, we make the best decisions we can) against an opponent that will choose the list (*i.e.*, the opponent will choose the hardest list possible)? Use the discussion of log base from Chapter 2 to help you justify your answer.

2. Does $f(n) \in 2^{O(\log_2(n))}$ imply that $f(n) \in O(n)$? If so explain why. If not, give a counterexample.

3. Does $f(n) \in 2^{o(\log_2(n))}$ imply that $f(n) \in O(n)$? If so explain why. If not, give a counterexample.

6.5 Answer key

1. We proceed with the same reasoning as before: we will start with all possible unsorted lists, and from those we want to reach a sorted result. When we consider the duplicate elements, we have $n!$ possible unsorted lists. We will have $d_1! \cdot d_2! \cdots$ allowed sorted lists (where d_1 is the number of elements with some equal value v_1); therefore, we are looking for the number of weighings on the scale that will take us from $n!$ to $d_1! \cdot d_2! \cdots$. Thus we need $\frac{n!}{d_1! \cdot d_2! \cdots}$. In the best-case scenario, weighing gives us three equal options, and thus if we employ the best strategy possible, we cannot do better than dividing the possible unsorted lists into thirds in each case; therefore, the depth of the tree will be

$$\log_3 \left(\frac{n!}{d_1! \cdot d_2! \cdots} \right) \in \Omega \left(log \left(\frac{n!}{d_1! \cdot d_2! \cdots} \right) \right).$$

   Here we can see that placing duplicate elements in the list will only improve the runtime (as compared to having no duplicate elements,

where the runtime was $\Omega(\log(n!))$). So a clever opponent will not choose any duplicate elements, and we see that we still have $\Omega(\log(n!))$.

2. No. Consider $g(n) = n^2 \notin O(n)$ with $\log(g(n)) = 2\log(n) \in O(\log(n))$.

3. Yes.
$$g(n) \in o(\log(n)) \to \lim_{n\to\infty} \frac{g(n)}{\log(n)} = 0.$$
We can think of this by splitting the $g(n)$ into a part that contains an $\log(n)$ term and a part that goes to zero:
$$\lim_{n\to\infty} \frac{g(n)}{\log(n)} = \lim_{n\to\infty} \frac{\log(n) \cdot t(n)}{\log(n)} = \lim_{n\to\infty} t(n) = 0;$$
therefore, we see that $t(n)$ must go to zero as $n \to \infty$.
$$f(n) = 2^{g(n)} = 2^{\log_2(n)\cdot t(n)} = \left(2^{\log_2(n)}\right)^{t(n)} = n^{t(n)}.$$
Because $t(n) \to 0$, we see that $t(n) < 1, n \gg 1$ and thus $n^{t(n)} < n^1, \gg 1$, and finally $f(n) = n^{t(n)} \in O(n)$.

Chapter 7

Recurrences and Memoization: The Fibonacci Sequence

The Fibonacci sequence occurs frequently in nature and has closed form $f(n) = f(n-1) + f(n-2)$, where $f(1) = 1$, $f(0) = 1$. This can be computed via a simple recursive implementation (Listing 7.1). On a large n, the recursive Fibonacci can be quite slow or can either run out of RAM or surpass `Python`'s allowed recursion limit[1]. But if we wanted to know the precise runtime of our recursive Fibonacci, this is difficult to say. To the untrained eye, it may look like a 2^n runtime, because it bifurcates at every non-leaf in the recursion; however, this is not correct because the call tree of a 2^n algorithm corresponds to a perfect binary tree, while the call trees from our recursive Fibonacci will be significantly deeper in some areas.

An iterative approch (Listing 7.2) can re-use previous computations and improve efficiency, computing the n^{th} Fibonacci number in $O(n)$. This iterative strategy is a type of "dynamic programming", a technique used to solve problems in a bottom-up fashion that reuses computations. Note that in calling `fib(100)` with our iterative method, it would only compute `fib(7)` a single time. In contrast, the naive recursive approach would compute `fib(7)` several times.

Listing 7.1: Recursive Fibonacci.

```
def fib(n):
  if n==0 or n==1:
      return 1
```

[1]The `Python` equivalent of a "stack overflow" error.

```
  return fib(n-1) + fib(n-2)
```

Listing 7.2: Iterative Fibonacci.

```
def fib(n):
  last_result = 1
  result = 1

  for i in xrange(n-1):
    next_result = last_result + result

    last_result = result
    result = next_result

  return result

N=100
for i in xrange(N):
  print fib(i)
```

7.1 Memoization

"Memoization" is the top-down counterpart to dynamic programming: rather than a programmer deliberately designing the algorithm to reuse computations in a bottom-up manner, memoization performs recursive calls, but caches previously computed answers. Like dynamic programming, memoization prevents redundant computations. Listing 7.3 shows a memoized implementation of the Fibonacci sequence.[2]

7.1.1 Graphical proof of runtime

To compute the runtime of the memoized variant, consider the call tree: As with the recursive version, `fib(n)` calls `fib(n-1)` (left subtree) and will *later* call `fib(n-2)` (right subtree)[3]. `fib(n-1)` is called first, and that will

[2]For simplicity, assume an $O(1)$ dictionary lookup for our cache. In the case of Fibonacci, we could always use an array of length n instead of a dictionary, should we need to.

[3]With the caveat that the memoized version also passes the cache as an additional parameter

call `fib(n-2)` (left subtree) and will *later* call `fib(n-3)` (right subtree). Proceeding in this manner, we can see that the base cases will be reached and then `fib(2)` will be computed and cached. That `fib(2)` was called by `fib(3)` (that `fib(2)` was the left subtree of its parent, and thus it was the `fib(n-1)` call, not the `fib(n-2)` call). `fib(3)` then calls `fib(1)` (which is the base case), to compute `fib(3)` and add it to the cache. That `fib(3)` was called by `fib(4)`, which will also call `fib(2)`; `fib(2)` is already in the cache. Note that *every* non-base case right subtree call will already be in the cache. Thus, when we draw the call tree, the right subtrees will all be leaves. The depth of the tree will be $n-1$ because that is the distance traveled before n decreases to either base cases ($n=1$ or $n=0$) through calls that decrease n by 1 in each recursion. Thus the total number of nodes in the call tree will be $\in Theta(n)$ and the runtime of the memoized Fibonacci function will be $\in \Theta(n)$.

7.1.2 Algebraic proof of runtime

Consider that each value can be added to the cache at most once, and since the work done in each of these recursive calls (an addition) costs $O(1)$, then the runtime is $\in O(n)$. Furthermore, `fib(i)` *must* be computed for $i \in \{0, 1, 2, \ldots, n\}$, because we need `fib(n-1)` and `fib(n-2)` to compute `fib(n)`. Hence, the runtime is $\in \Omega(n)$. Thus we verify what we saw above: the runtime of the memoized Fibonacci function will be $\in \Theta(n)$.

Listing 7.3: Memoized Fibonacci.

```
# if no cache argument is given, start with an empty cache:
def fib(n, cache={}):
  if n==0 or n==1:
    return 1

  if n not in cache:
    cache[n] = fib(n-1,cache) + fib(n-2,cache)

  # i must be in cache now:
  return cache[n]

N=100
print 'with empty cache every time (slower):'
for i in xrange(N):
  print fib(i)
```

```
print

print 'keeping previous work (faster):'
cache={}
for i in xrange(N):
  print fib(i, cache)
```

Furthermore, we can reuse the cache in subsequent recursive calls. If we do this, it will ensure computing every Fibonacci number from 0 to n will cost $O(n)$ in total (regardless of the order in which we compute them). Equivalently, it means that if we compute every Fibonacci number from 0 to n (regardless of the order in which we compute them), the amortized runtime per call will be $\widetilde{O}(1)$.

7.2 Recurrence closed forms and "eigendecomposition"

The question then arises: can we do better than our memoized version? Perhaps, but we need to step back and consider this problem from a mathematical perspective. First, let's write the Fibonacci computation using linear algebra:

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) \\ &= \begin{bmatrix} 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix}. \end{aligned}$$

This does nothing useful yet; it simply restates this in a standard mathematical form, revealing that multiplication with the vector [1 1] advances two neighboring Fibonacci numbers to compute the following Fibonacci number. But if we want to chain this rule together and use it iteratively, we have a problem: our function takes

$$\begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix}$$

a length-2 vector, as an input, but it produces a single numeric result. For this reason, as stated, the above mathematical formalism cannot be easily applied multiple times.

We can easily adapt this linear algebra formulation to produce a length-2 output: feeding in the previous two Fibonacci numbers $f(n-1)$ and $f(n-2)$ should yield the next pair $f(n)$ and $f(n-1)$:

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix}.$$

The "characteristic matrix" of the Fibonacci recurrence is

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

We can see that we start with base case values

$$\begin{bmatrix} f(1) \\ f(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix};$$

therefore, we can compute the n^{th} via

$$\begin{aligned} & \underbrace{A \cdot A \cdots A}_{n-1 \text{ terms}} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ = \quad & A^{n-1} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix}. \end{aligned}$$

This does not yet help us compute our Fibonacci values faster than $O(n)$, but we have no moved into a more theoretical domain where useful answers may exist.

Passing a vector through a square matrix is a bit like shooting an arrow into a storm: it may accelerate, decelerate, turn, or reverse (or some combination) as it goes through the storm, and a new arrow will be shot out. There are special directions where shooting the arrow in will only accelerate or decelerate it, but not turn it. These are called the "eigenvectors"[4]. Each eigenvector has a corresponding eigenvalue, the amount by which the vector is stretched after being shot through the storm. For example, an eigenvector v_1 with paired eigenvalue λ_1 will mean that $Av_1 = \lambda_1 v_1$, *i.e.*, that shooting through our little storm stretched the vector by constant λ_1.

[4]From the German "eigen" meaning "self"

Since we have a two-dimensional problem, we need at most two eigenvectors to fully describe the space (they are like axes). For this reason, we can think of our initial

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

vector as being composed of some amounts from each of those two eigenvector ingredients. Thus, we have

$$\begin{aligned} \begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} &= A^{n-1} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ &= A^{n-1} \cdot (c_1 v_1 + c_2 v_2) \\ &= c_1 \lambda_1^{n-1} v_1 + c_2 \lambda_2^{n-1} v_2 \\ f(n) &= c_1 \lambda_1^{n-1} v_1[0] + c_2 \lambda_2^{n-1} v_2[0]. \end{aligned}$$

From this we see that this would result in a two-term exponential sequence with four free parameters, d_1, λ_1, d_2, and λ_2 (where $d_1 = c_1 \cdot v_1[0]$ and $d_2 = c_2 \cdot v_2[0]$). If we fit the free parameters that would produce $f(0) = 1$, $f(1) = 1$, $f(2) = 2$, $f(3) = 3$, we find

$$\begin{aligned} f(n) &= \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{n+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{n+1} \right) \\ &= \frac{1}{\sqrt{5}} \left(1.618\ldots^{n+1} - (-0.6180\ldots)^{n+1} \right) \end{aligned}$$

Interestingly, the larger of the eigenvalues, $\frac{1+\sqrt{5}}{2}$ equals the golden ratio $\phi \approx 1.618$ (we can also compute these numerically via `numpy`, as shown in Listing 7.4). This is our Fibonacci closed form, which runs in $\Theta(1)$ if we have access to an $O(1)$ function for computing a^b for arbitrary a and b. An implementation of this Fibonacci closed form can be seen in Listing 7.5.

Listing 7.4: Numerically computing the closed form from the characteristic matrix via `numpy`.

```
import numpy

# Fibonacci characteristic matrix:
A=numpy.matrix([[1,1],[1,0]])

# Eigenvalues:
```

```
lambda1, lambda2 = numpy.linalg.eigvals(A)

# Solve for coefficeints d1,d2:
# z(t) = d1*lambda1**(t-1) + d2*lambda2**(t-1):
# <--> M*[[d1],[d2]] = [1,1]
M=numpy.matrix([[lambda1**-1,lambda2**-1],[lambda1**0,lambda2**0]])
d = numpy.linalg.solve(M, [[1],[1]])
d1,d2 = d.T[0]

def z(t):
  return d1*lambda1**(t-1) + d2*lambda2**(t-1)

for t in xrange(10):
  print z(t)
```

When we do not have access to an $O(1)$ function for computing a^b, we can compute our Fibonacci function in $\Theta(\log(n))$ using the recurrence

$$a^b = \begin{cases} \left(a^{\frac{b}{2}}\right)^2 & b \text{ is even} \\ a \cdot \left(a^{\frac{b-1}{2}}\right)^2 & \text{else.} \end{cases}$$

The recurrence for a^b is easily found by memoizing while computing a^b.

Listing 7.5: Closed form of Fibonacci sequence in `Python`.

```
import numpy as np

def fib(n):
  root_of_5 = np.sqrt(5)
  lambda_1=(1+root_of_5)/2.0
  lambda_2=(1-root_of_5)/2.0
  return 1/root_of_5 * ( lambda_1**(n+1) - lambda_2**(n+1) )

N=100
for i in xrange(N):
  print i, fib(i)
```

7.3 The runtime of the naive recursive Fibonacci

Each 1 added to the result can come only from a leaf (*i.e.*, base cases) in the call tree. If we only include these leaves in the runtime (excluding any of the other nodes in the call tree or additions performed), we get

$$\begin{aligned} \text{runtime} &\geq \text{runtime from leaves} \\ &\geq \text{number of leaves} \\ &\in \Omega(\phi^n). \end{aligned}$$

For that reason we can see that the runtime of Fibonacci is bounded below: $f(n) \in \Omega(\phi^n) \subset \Omega(1.618^n)$ (because $\phi > 1.618$).

7.4 Generic memoization

Connoisseurs of computer science will observe the above trajectory from recurrence to closed form and will ask, "Why don't we just use the recurrence to identify which famous sequence this is, and then look up its closed form?" The answer to this is simple: what would we do for a sequence that isn't famous?

Likewise, we should resist the urge to always insist on a closed form: we cannot always construct a closed form using known techniques. Indeed, there are recurrent functions that seem to always terminate, but where this has not yet been proven, in spite of real effort[5]. For this reason, memoization can still be a suprisingly vital tool in its own right.

Listing 7.6 shows how we can make a generic "decorator" in `Python`, which will can be used to adapt new functions to memoized forms. For example, consider

$$g(a,b) = \begin{cases} 1 & a \leq 0 \text{ or } b \leq 0 \\ g(a-2,b+1) + g(a+1,b-2) + g(a-1,b-1) & \text{else.} \end{cases}$$

Without computing a closed form, we can still compute this function without fear of recomputing the same values again and again (as we did with our initial naive Fibonacci implementation).

[5] See the "hailstone sequence".

Listing 7.6: Generic memoization using a "decorator" in `Python`.

```
# class used as function "decorator":
class Memoized:
  def __init__(self, function):
    self._function = function
    self._cache = {}
  def __call__(self, *args):
    if args not in self._cache:
      # not in the cache: call the function and store the result in
      # the cache
      self._cache[args] = self._function(*args)

    # the result must now be in the cache:
    return self._cache[args]

@Memoized
def fib(n):
  if n==0 or n==1:
      return 1
  return fib(n-1) + fib(n-2)

print fib(10)


@Memoized
def g(a,b):
  if a<=0 or b<=0:
      return 1
  return g(a-2,b+1) + g(a+1,b-2) + g(a-1,b-1)

print g(10,3)
```

7.5 Practice and discussion questions

1. You work as a researcher combatting the zombie virus that is infecting American cities. At day $t = 0$, patient zero moves to Missoula from a large metropolis. In each day, a zombie has a 20% chance of biting a non-zombie, and so the rate of infection grows with the number of infected. Furthermore, you find that those newly created zombies tend to meet up with one another and produce zombie offspring. Where $z(t)$

is the number of zombies on day t, your model predicts

$$z(t) = \begin{cases} 1 & t = 0 \text{ or } t = 1 \text{ or } t = 2 \\ 1.2z(t-1) + 0.2\left(z(t-2) + z(t-3)\right) & \text{else.} \end{cases}$$

(We will ignore the fact that there will be fewer people to infect as the infection progresses.)

(a) Write a memoized implementation of z in `python`.

(b) The official Missoula website estimates the population at just over 69k people– after how many days t will $z(t) \geq 69000$?

(c) Find the characteristic matrix of the recurrence z.

(d) Find its eigenvalues $\lambda_1, \lambda_2, \lambda_3$. Note that even though our sequence is on the reals, these eigenvalues may be complex values.

(e) Find the closed form of $z(t)$. You may use numeric approximations of constants computed (round to four significant digits).

(f) Repeat your analysis of how many days it will take before $z(t) > 69000$, but this time use your closed form for $z(t)$. Your result should match what you computed with the memoized form.

2. Let

$$h(x, y, z) = \begin{cases} 1 & x \leq y \text{ or } y \leq z \text{ or } z \leq 5 \\ h(\frac{x}{2}, x + y + z, y) & x \text{ is even} \\ h(x - 1, y, z) & \text{else} \end{cases}$$

(a) Write a recursive, memoized implementation of `h` in `Python` code.

(b) Could we use the methods taught in this chapter to compute the closed form of h? If so, compute the closed form. If not, explain why.

7.6 Answer key

```
import numpy

class Memoized:
  def __init__(self, function):
    self._function = function
```

```
    self._cache = {}
  def __call__(self, *args):
    if args not in self._cache:
      # not in the cache: call the function and store the result in
      # the cache
      self._cache[args] = self._function(*args)

    # the result must now be in the cache:
    return self._cache[args]

# 1a)
@Memoized
def z_mem(t):
  if t in (0,1,2):
    return 1
  return 1.2*z_mem(t-1) + 0.2*z_mem(t-2) + 0.2*z_mem(t-3)

# 1b)
t=0
while z_mem(t) < 69000:
  print t, z_mem(t)
  t += 1

print 'Crossed population of Missoula on day', t, 'with this many
    zombies:', z_mem(t)

# 1c)
A=numpy.matrix([[1.2,0.2,0.2],[1,0,0],[0,1,0]])

# 1d)
lambda1, lambda2, lambda3 = numpy.linalg.eigvals(A)

# 1e)
# solve z(0)=1, z(1)=1, z(2)=1 for d1,d2,d3:
M=numpy.matrix([[lambda1**-1,lambda2**-1,lambda3**-1],
                [lambda1**0,lambda2**0,lambda3**0],
                [lambda1**1,lambda2**1,lambda3**1]])
d = numpy.linalg.solve(M, [[1],[1],[1]])
d1,d2,d3 = d.T[0]

def z(t):
  return d1*lambda1**(t-1) + d2*lambda2**(t-1) + d3*lambda3**(t-1)

# test our closed form against memoized version:
for t in range(10):
```

```
  print t, z_mem(t), z(t)

# 1f)
t=0
while z(t) < 69000:
  print t, z(t)
  t += 1

print 'Crossed population of Missoula on day', t, 'with this many
    zombies:', z(t)

# 2a)

@Memoized
def h(x,y,z):
  if x <= y or y <= z or z <= 5:
    return 1
  if x % 2 == 0:
    return h(x/2,x+y+z,y)
  return h(x-1,y,z)

print h(500,200,100)

# 2b)

# We can't use techniques learned here to compute a closed form of h;
# we only learned how to do
# f(t) = a*f(t-1) + b*f(t-2) + ... for constants a, b, ...
# with base case f(0)=r0, f(1)=r1, ... for constants r0, r1, ...
```

Chapter 8

Recurrences: The Master Theorem

In Chapter 7, we discussed recurrences that were used to compute a single *value*, such as the value of the n^{th} Fibonacci number. Recurrences are also used so describe the runtimes of divide-and-conquer algorithms.

Divide-and-conquer algorithms, like merge sort from Chapter 4, are so important that special tools have been developed for creating bounds from their runtime recurrences. A great example of these tools is the "master theorem".

Consider the family of runtime recurrences $r(n) = a \cdot r(\frac{n}{b}) + f(n)$ where a and b are constants. This corresponds to a divide-and-conquer method where each invocation calls a recursions of size $\frac{n}{b}$ and where each invocation also performs $f(n)$ steps of actual work. For example, we saw that merge sort has a runtime recurrence $r(n) = 2r(\frac{n}{2}) + n$, because it reduces to two recursive calls, each with half the size, and a single pass to merge the sorted results from each half.

8.1 Call trees and the master theorem

Because the problem size is divided by b in each recursion, the depth d of the call tree will clearly be

$$d = \log_b(n) = \frac{\log_2(n)}{\log_2(b)}.$$

Each instance will spawn a recursive calls, and so the number of leaves ℓ will be

$$a^d = a^{\frac{\log_2(n)}{\log_2(b)}}.$$

We can rearrange this expression by observing

$$\begin{aligned} \log_2(a^d) &= d\log_2(a) \\ &= \frac{\log_2(n)\log_2(a)}{\log_2(b)}. \end{aligned}$$

By exponentiating again, we can see

$$\begin{aligned} \ell = a^d &= 2^{\frac{\log_2(n)\log_2(a)}{\log_2(b)}} \\ &= \left(2^{\log_2(n)}\right)^{\frac{\log_2(a)}{\log_2(b)}} \\ &= n^{\frac{\log_2(a)}{\log_2(b)}} \\ &= n^{\log_b(a)}. \end{aligned}$$

Thus we see that the number of leaves ℓ in the call tree will be $\ell = n^{\log_b(a)}$.

The master theorem revolves around three cases: the case where the leaves dominate the computation, the case where the root dominates the computation, and the case where neither dominate the other and thus every node in the call tree is asymptotically significant.

8.2 "Leaf-heavy" divide and conquer

In the case of leaf-heavy divide-and-conquer algorithms, the cost from the sheer number of leaves ℓ is substantially larger than the cost of the work done at the root. For example, Strassen matrix multiplication (discussed in Chapter 12) reduces matrix multiplication of two $n \times n$ matrices to 7 matrix multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and $\Theta(n^2)$ postprocessing step. Here, the cost of work done at the root is $\Theta(n^2)$, but the number of leaves is $n^{\log_2(7)} \approx n^{2.807}$. Even if each leaf takes a trivial (but nonzero) amount of processing time, the number of leaves ℓ dwarfs the work done at the root: $n^2 \in O(n^{\log_2(7)-\epsilon})$.[1] In general, the leaf-heavy case applies when $f(n) \in O(n^{\log_b(a)-\epsilon})$.

[1]Note that $f(n) \in O\left(n^{c-\epsilon}\right)$ is a stronger form of the statement $f(n) \in o(n^c)$; $f(n) \in O\left(n^{c-\epsilon}\right)$ implies $f(n) \in o(n^c)$, but $f(n) \in o(n^c)$ does not necessarily imply

When we sum up the total computation cost over all recursions, we see

$$\begin{aligned}
r(n) &= \sum_{i=0}^{d} a^i \cdot f\left(\frac{n}{b^i}\right) \\
&< c\sum_{i=0}^{d} a^i \cdot \left(\frac{n}{b^i}\right)^{\log_b(a)-\epsilon}, \ n \gg 1 \\
&\propto \sum_{i=0}^{d} a^i \cdot \frac{n^{\log_b(a)-\epsilon}}{b^{i\cdot(\log_b(a)-\epsilon)}}, \ n \gg 1 \\
&= \sum_{i=0}^{d} a^i \cdot \frac{n^{\log_b(a)-\epsilon}}{\left(b^{\log_b(a)-\epsilon}\right)^i}, \ n \gg 1 \\
&= \sum_{i=0}^{d} a^i \cdot \frac{n^{\log_b(a)} \cdot n^{-\epsilon}}{\left(a \cdot b^{-\epsilon}\right)^i}, \ n \gg 1 \\
&= \sum_{i=0}^{d} \frac{a^i}{a^i} b^{\epsilon\cdot i} \cdot n^{\log_b(a)} \cdot n^{-\epsilon}, \ n \gg 1 \\
&= n^{\log_b(a)} \cdot n^{-\epsilon} \sum_{i=0}^{d} \left(b^{\epsilon}\right)^i, \ n \gg 1 \\
&= n^{\log_b(a)} \cdot n^{-\epsilon} \cdot b^{\epsilon} \frac{\left(b^{\epsilon}\right)^d - 1}{b^{\epsilon} - 1}, \ n \gg 1 \\
&= n^{\log_b(a)} \cdot n^{-\epsilon} \cdot b^{\epsilon} \frac{\left(b^{\log_b(n)}\right)^{\epsilon} - 1}{b^{\epsilon} - 1}, \ n \gg 1 \\
&= n^{\log_b(a)} \cdot n^{-\epsilon} \cdot b^{\epsilon} \frac{n^{\epsilon} - 1}{b^{\epsilon} - 1}, \ n \gg 1 \\
&= n^{\log_b(a)} \cdot b^{\epsilon} \frac{1 - n^{-\epsilon}}{b^{\epsilon} - 1}, \ n \gg 1 \\
&< n^{\log_b(a)} \cdot b^{\epsilon} \frac{1}{b^{\epsilon} - 1}, \ n \gg 1 \\
&\propto n^{\log_b(a)}, \ n \gg 1,
\end{aligned}$$

where c is chosen as an arbitrary constant to satisfy the definition of $O(\cdot)$ and

$f(n) \in O\left(n^{c-\epsilon}\right)$. For example, consider the case where $f(n) = \frac{n^2}{\log(n)}$; $f(n) \in o(n^2)$, but $f(n) \notin O(n^{2-\epsilon})$.

all proportionality constants are nonnegative. Thus, the runtime is $r(n) \in O\left(n^{\log_b(a)}\right)$. Furthermore, because we visit each of the $n^{\log_b(a)}$ leaves, the runtime must be $r(n) \in \Omega\left(n^{\log_b(a)}\right)$; therefore, the runtime of the leaf-heavy case is $r(n) \in \Theta\left(n^{\log_b(a)}\right)$.

8.3 "Root-heavy" divide and conquer

In root-heavy divide and conquer, the cost of visiting the root dwarfs the number of leaves. For example, a recurrence of the form $r(n) = 2r\left(\frac{n}{2}\right) + n^2$ will result in $n^{\log_2(2)} = n$ leaves but visiting the root alone will cost n^2. The cost of visiting the root is $n^2 \in \Omega(n^2)$, and so the total cost (which includes the visit to the root) must be $\in \Omega(n^2)$.

More generally, when $f(n) \in \Omega(n^{\log_b(a)+\epsilon})$ the total cost $r(n) \in \Omega(f(n))$; however, unlike the leaf-heavy case, this does not yield a corresponding $O(\cdot)$ upper bound. For this reason, to make good use of the root-heavy case, we add an additional constraint known as the "regularity condition":

$$a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n),\ n \gg 1,$$

where $c < 1$. Under this condition, we see that for $n \gg 1$,

$$a^i \cdot f\left(\frac{n}{b^i}\right) \leq c \cdot a^{i-1} \cdot f\left(\frac{n}{b^{i-1}}\right) \leq \cdots \leq c^i \cdot f(n).$$

This creates an upper bound on the total runtime:

$$\begin{aligned}
r(n) &= \sum_{i=0}^{d} a^i \cdot f\left(\frac{n}{b^i}\right) \\
&\leq \sum_{i=0}^{d} c^i \cdot f(n),\ n \gg 1 \\
&= f(n) \sum_{i=0}^{d} c^i,\ n \gg 1 \\
&< f(n) \frac{1}{1-c},\ n \gg 1 \\
&\propto f(n),\ n \gg 1,
\end{aligned}$$

where the proportionality constants are nonnegative. Thus, under the regularity condition of the root-heavy case, we see that $r(n)$ is bounded above $r(n) \in O(f(n))$. Because we have already shown $r(n) \in \Omega(f(n))$ for the root-heavy case, we see that $r(n) \in \Theta(f(n))$.

8.3.1 A note on the meaning of regularity

Regularity means that the layers will do an exponentially decaying amount of work.

A root-heavy example where we have regularity is

$$r(n) = 2r\left(\frac{n}{4}\right) + 8n^2.$$

We see that this recurrence is root heavy because $8n^2 \in \Omega\left(n^{\log_4(2)+\epsilon}\right) = \Omega\left(n^{\frac{1}{2}+\epsilon}\right)$. We see that regularity is met because $2r\left(\frac{n}{4}\right) = 2 \cdot 8\left(\frac{n}{4}\right)^2 = n^2$, and $n^2 \leq \frac{1}{8}f(n)$, which satisfies with $c = \frac{1}{8}$.

A root-heavy example where we do not have regularity can be constructed by making an $f(n)$ that is not monotonic. For example, $r(n) = r\left(\frac{n}{2}\right) + n \cdot (2 + \cos(\log_2(n) \cdot pi))$ does not have regularity because the sum of all $f(n)$ terms does not always decay in the next layer. Consider that $\cos(\log_2(n) \cdot \pi)$ will oscillate in sign as n is divided by 2 in each recursion. Thus, we may run $f(n) = 3n$ at the root and then $f\left(\frac{n}{2}\right) = \frac{n}{2}$ in the next iteration (which does appear to decay exponentially); however, in the subsequent recursion, we would have $f\left(\frac{n}{4}\right) = 3\frac{n}{4}$, which is actually larger again. In short, such non-monotonic f could be dangerous because the fact that the cost of the root is higher than the cost of the leaves does not guarantee that the cost of the root is the highest cost in computing $r(n)$; *i.e.*, the highest cost may occur somewhere in between the root and the leaves.

Note that even though the master theorem is uncertain about this particular case, we could make an upper bound for $r(n)$ as $r\left(\frac{n}{2}\right) + n$, which *would* have regularity: $a \cdot r\left(\frac{n}{b}\right) = 1 \cdot \frac{n}{2} = \frac{n}{2} = \frac{1}{2} \cdot n$, which satisfies regularity with $c = \frac{1}{2}$.

8.4 "Root-leaf-balanced" divide and conquer

In the balanced case, neither the root nor the leaves dominate the runtime. Intuitively, we can see that in this case the next level of the tree $i + 1$ must

not take substantially more (where substantially more means $\in \Omega(n^{c+\epsilon})$) nor substantially less (where substantially less means $\in O(n^{c-\epsilon})$) runtime than level i in the tree. One way to achieve this is to have $f(n) \in \Theta(n^{\log_b(a)})$; however, we can be more general than this and permit $f(n) \in \Theta(n^{\log_b(a)}(\log(n))^k)$ for some constant k, because any constant number of logarithmic terms are not enough to overcome a $\pm\epsilon$ change to the polynomial power and thus will cause neither a leaf-heavy nor a root-heavy outcome as defined above.

As in the leaf-heavy case, we can sum up the actual work done during f in all recursions. This easily proves the upper bound of the runtime:

$$\begin{aligned}
r(n) &= \sum_{i=0}^{d} a^i \cdot f\left(\frac{n}{b^i}\right) \\
&< c_1 \sum_{i=0}^{d} a^i \cdot \left(\frac{n}{b^i}\right)^{\log_b(a)} \left(\log\left(\frac{n}{b^i}\right)\right)^k,\ n \gg 1 \\
&\propto \sum_{i=0}^{d} a^i \cdot \frac{n^{\log_b(a)}}{(b^i)^{\log_b(a)}} \left(\log\left(\frac{n}{b^i}\right)\right)^k,\ n \gg 1 \\
&= n^{\log_b(a)} \sum_{i=0}^{d} a^i \cdot \frac{1}{\left(b^{\log_b(a)}\right)^i} \left(\log\left(\frac{n}{b^i}\right)\right)^k,\ n \gg 1 \\
&= n^{\log_b(a)} \sum_{i=0}^{d} \frac{a^i}{a^i} \left(\log\left(\frac{n}{b^i}\right)\right)^k,\ n \gg 1 \\
&= n^{\log_b(a)} \sum_{i=0}^{d} \left(\log\left(\frac{n}{b^i}\right)\right)^k,\ n \gg 1 \\
&< n^{\log_b(a)} \sum_{i=0}^{d} (\log(n))^k,\ n \gg 1 \\
&= n^{\log_b(a)} (\log(n))^k \cdot d,\ n \gg 1 \\
&= n^{\log_b(a)} (\log(n))^k \cdot \log_b(n),\ n \gg 1 \\
&\in O(n^{\log_b(a)} (\log(n))^{k+1},\ n \gg 1.
\end{aligned}$$

The lower bound is a little trickier, but we begin in the same manner:

$$\begin{aligned}
r(n) &= \sum_{i=0}^{d} a^i \cdot f\left(\frac{n}{b^i}\right) \\
&> c_2 \sum_{i=0}^{d} a^i \cdot \left(\frac{n}{b^i}\right)^{\log_b(a)} \left(\log\left(\frac{n}{b^i}\right)\right)^k,\ n \gg 1 \\
&\propto \sum_{i=0}^{d} a^i \cdot \frac{n^{\log_b(a)}}{(b^i)^{\log_b(a)}} \left(\log\left(\frac{n}{b^i}\right)\right)^k,\ n \gg 1 \\
&= n^{\log_b(a)} \sum_{i=0}^{d} a^i \cdot \frac{1}{\left(b^{\log_b(a)}\right)^i} \left(\log\left(\frac{n}{b^i}\right)\right)^k,\ n \gg 1 \\
&= n^{\log_b(a)} \sum_{i=0}^{d} \frac{a^i}{a^i} \left(\log\left(\frac{n}{b^i}\right)\right)^k,\ n \gg 1 \\
&= n^{\log_b(a)} \sum_{i=0}^{d} \left(\log\left(\frac{n}{b^i}\right)\right)^k,\ n \gg 1 \\
&= n^{\log_b(a)} \sum_{i=0}^{d} \left(\log(n) - i \cdot \log(b)\right)^k,\ n \gg 1.
\end{aligned}$$

We will try to find a lower bound for the sum. Consider the fact that

$$\sum_{i=0}^{d} \left(\log(n) - i \cdot \log(b)\right)^k$$

is nonnegative and strictly descending. Thus we can find a lower bound as we did for $\log(n!)$ in Chapter 6 by zeroing out the smaller terms and then

replacing each remaining term with the smallest term remaining.

$$\begin{aligned}
\sum_{i=0}^{d} (\log(n) - i \cdot \log(b))^k &= \sum_{i=0}^{\log_b(n)} (\log(n) - i \cdot \log(b))^k \\
&> \sum_{i=0}^{\frac{\log_b(n)}{2}+1} (\log(n) - i \cdot \log(b))^k \\
&> \left(\frac{\log_b(n)}{2} + 1\right) \cdot \left(\log(n) - \frac{\log_b(n)}{2} \cdot \log(b)\right)^k \\
&= \left(\frac{\log_b(n)}{2} + 1\right) \cdot \left(\log(n) - \frac{\log(n)}{2}\right)^k \\
&= \left(\frac{\log_b(n)}{2} + 1\right) \cdot \left(\frac{\log(n)}{2}\right)^k.
\end{aligned}$$

Note that the 2^k in the denominator is a constant; it may be large, but 2 to any constant is also a constant; therefore we have

$$\sum_{i=0}^{d} (\log(n) - i \cdot \log(b))^k \in \Omega\left((\log(n))^{k+1}\right).$$

Thus

$$r(n) \in \Omega\left(n^{\log_b(a)} \cdot (\log(n))^{k+1}\right),$$

and together with our $O(\cdot)$ result above, this implies that

$$r(n) \in \Theta\left(n^{\log_b(a)} \cdot (\log(n))^{k+1}\right).$$

8.5 Practice and discussion questions

1. What are the three cases for the master theorem and what are their requirements? For the root-heavy case, consider the case with and without regularity. Also name the strictest implications made in each case.

   For each divide-and-conquer runtime recurrence, apply the master theorem if it applies. If the root-heavy case is used, specify the runtime implications with and without regularity.

2. $r(n) = 3r\left(\frac{n}{2}\right) + 3n$
3. $r(n) = 3r\left(\frac{n}{2}\right) + 2n^2$
4. $r(n) = 2r\left(\frac{n}{2}\right) + n$
5. $r(n) = 2r\left(\frac{n}{2}\right) + n\log(n)$
6. $r(n) = 2r\left(\frac{n}{2}\right) + 2n(\log(n))^{100}$
7. $r(n) = r\left(\frac{n}{2}\right) + 7n$
8. $r(n) = r\left(\frac{n}{2}\right) + n$
9. $r(n) = r\left(\frac{n}{8}\right) + 3n$
10. $r(n) = \log(n)r\left(\frac{n}{2}\right) + 7\sqrt{n}$
11. $r(n) = 3r\left(\frac{n}{2}\right) + 3n$
12. $r(n) = 7r\left(\frac{n}{2}\right) + 6n^2$
13. $r(n) = 10r\left(\frac{n}{4}\right) + 16n^2$
14. $r(n) = r\left(\frac{n}{2}\right) + 2\sqrt{n}$
15. $r(n) = 9r\left(\frac{n}{4}\right) + 7n^2$
16. $r(n) = 9r\left(\frac{n}{2}\right) + 8n^2$
17. $r(n) = 9r\left(\frac{n}{3}\right) + 9n^2$
18. $r(n) = 9r\left(\frac{n}{3}\right) + 8n^2\log(n)$
19. $r(n) = 9.001r\left(\frac{n}{3}\right) + 14n^2\log(n)$
20. $r(n) = 3r\left(\frac{n}{9}\right) + 6\sqrt{n}$
21. $r(n) = 3r\left(\frac{n}{9}\right) + 2\sqrt{n}\log(n)$
22. $r(n) = 3r\left(\frac{n}{9.001}\right) + 4\sqrt{n}\log(n)$
23. $r(n) = 2r\left(\frac{n}{\log(n)}\right) + 9n$

24. $r(n) = 2r\left(\frac{n}{2}\right) + nk\log(nk)$, where k is constant.

25. For the recurrence in the previous question, draw the call tree and compute the runtime of that divide-and-conquer algorithm where k is not constant (by summing over every layer rather than using the master theorem). What is the runtime with respect to n and k?

8.6 Answer key

1.
   - Leaf-heavy: $f(n) \in O\left(n^{\log_b(a)-\epsilon}\right) \to r(n) \in \Theta\left(n^{\log_b(a)}\right)$
   - Root-leaf balanced: $f(n) \in \Theta\left(n^{\log_b(a)}(\log(n))^k\right) \to r(n) \in \Theta\left(n^{\log_b(a)}(\log(n))^{k+1}\right)$
   - Root-heavy: $f(n) \in \Omega\left(n^{\log_b(a)+\epsilon}\right) \to r(n) \in \Omega(f(n))$. With regularity, $\to r(n) \in \Theta(f(n))$.

2. $r(n) = 3r\left(\frac{n}{2}\right) + 3n$ is leaf heavy, because $\log_2(3) > 1$; therefore, $r(n) \in \Theta\left(n^{\log_2(3)}\right)$.

3. $r(n) = 3r\left(\frac{n}{2}\right) + 2n^2$ is root heavy, because $\log_2(3) < 2$; therefore, $r(n) \in \Omega(n^2)$. With regularity, $r(n) \in \Theta(n^2)$.

4. $r(n) = 2r\left(\frac{n}{2}\right) + n$ is root-leaf balanced, because $\log_2(2) = 1$; therefore, $r(n) \in \Theta\left(n\log(n)\right)$.

5. $r(n) = 2r\left(\frac{n}{2}\right) + n\log(n)$ is root-leaf balanced, because $\log_2(2) = 1$; therefore, $r(n) \in \Theta\left(n\log(n)\log(n)\right)$.

6. $r(n) = 2r\left(\frac{n}{2}\right) + 2n(\log(n))^{100}$ is root-leaf balanced, because $\log_2(2) = 1$; therefore, $r(n) \in \Theta\left(n(\log(n))^{101}\right)$.

7. $r(n) = r\left(\frac{n}{2}\right) + 7n$ is root heavy, because $\log_2(1) < 1$; therefore, $r(n) \in \Omega(n)$. With regularity, $r(n) \in \Theta(n)$.

8. $r(n) = r\left(\frac{n}{2}\right) + n$ has the same answer as the previous question; changing $f(n) = 7n$ to $f(n) = n$ changes nothing.

9. $r(n) = r\left(\frac{n}{8}\right) + 3n$ is likewise root heavy: $\log_8(1) = 0 < 1$; therefore, $r(n) \in \Omega(n)$. With regularity, $r(n) \in \Theta(n)$.

10. $r(n) = \log(n)r\left(\frac{n}{2}\right) + 7\sqrt{n}$ does not admit a result under the master theorem; a is not a constant.

11. $r(n) = 3r\left(\frac{n}{2}\right) + 3n$ is leaf heavy, because $\log_2(3) > 1$; therefore, $r(n) \in \Theta\left(n^{\log_2(3)}\right)$.

12. $r(n) = 7r\left(\frac{n}{2}\right) + 6n^2$ is leaf heavy, because $\log_2(7) > 2$; therefore, $r(n) \in \Theta\left(n^{\log_2(7)}\right)$.

13. $r(n) = 10r\left(\frac{n}{4}\right) + 16n^2$ is root heavy, because $\log_4(10) < 2$; therefore, $r(n) \in \Omega(n^2)$. With regularity, $r(n) \in \Theta(n^2)$.

14. $r(n) = r\left(\frac{n}{2}\right) + 2\sqrt{n}$ is root heavy, because $\log_2(1) < \frac{1}{2}$ (note that $\frac{1}{2}$ is the power of $\sqrt{n}$); therefore, $r(n) \in \Omega(\sqrt{n})$. With regularity, $r(n) \in \Theta(\sqrt{n})$.

15. $r(n) = 9r\left(\frac{n}{4}\right) + 7n^2$ is root heavy, because $\log_4(9) < 2$; therefore, $r(n) \in \Omega(n^2)$. With regularity, $r(n) \in \Theta(n^2)$.

16. $r(n) = 9r\left(\frac{n}{2}\right) + 8n^2$ is leaf heavy, because $\log_2(9) > 2$; therefore, $r(n) \in \Theta(n^{\log_2(9)})$.

17. $r(n) = 9r\left(\frac{n}{3}\right) + 9n^2$ is root-leaf balanced, because $\log_3(9) = 2$; therefore, $r(n) \in \Theta\left(n^2 \log(n)\right)$.

18. $r(n) = 9r\left(\frac{n}{3}\right) + 8n^2\log(n)$ is root-leaf balanced for the same reason as the question above; therefore, $r(n) \in \Theta\left(n^2 \log(n) \log(n)\right)$.

19. $r(n) = 9.001r\left(\frac{n}{3}\right) + 14n^2\log(n)$ is leaf heavy, because $\log_3(9.001) > 2$; therefore, $r(n) \in \Theta\left(n^{\log_2(9.001)}\right)$.

20. $r(n) = 3r\left(\frac{n}{9}\right) + 6\sqrt{n}$ is root-leaf balanced, because $\log_9(3) = \frac{1}{2}$; therefore, $r(n) \in \Theta(\sqrt{n} \cdot \log(n))$.

21. $r(n) = 3r\left(\frac{n}{9}\right) + 2\sqrt{n}\log(n)$ is root-leaf balanced for the same reason as the question above; therefore, $r(n) \in \Theta(\sqrt{n}\log(n)\log(n))$.

22. $r(n) = 3r\left(\frac{n}{9.001}\right) + 4\sqrt{n}\log(n)$ is root heavy; $\log_{9.001}(3) < \frac{1}{2}$; therefore, $r(n) \in \Omega(\sqrt{(n)}\log(n))$. With regularity, $r(n) \in \Theta(\sqrt{n}\log(n))$.

23. $r(n) = 2r\left(\frac{n}{\log(n)}\right) + 9n$ does not admit a result under the master theorem; b is not a constant.

24. When k is a constant that does not depend on n, $r(n) = 2r\left(\frac{n}{2}\right) + nk\log(nk)$ has an $f(n) \in O(n\log(n))$. This is root-leaf balanced, because $\log_2(2) = 1$; therefore, $r(n) \in \Theta(n\log(n)\log(n))$.

25. When k is not a constant (*i.e.*, when k may depend on n), $r(n) = 2r\left(\frac{n}{2}\right) + nk\log(nk)$ will lead to a runtime:

$$\begin{aligned}
r(n) &= \sum_{i=0}^{d} 2^i \frac{nk}{2^i} \log\left(\frac{nk}{2^i}\right) \\
&= \sum_{i=0}^{d} nk \cdot \left(\log(nk) - \log(2^i)\right) \\
&= \sum_{i=0}^{d} nk \cdot (\log(nk) - i) \\
&= nk \cdot \left(\sum_{i=0}^{d} \log(nk) - \sum_{i=0}^{d} i\right) \\
&= nk \cdot \left(d\log(nk) - \frac{d\cdot(d+1)}{2}\right) \\
&= nk \cdot \left(\log_b(n)\log(nk) - \frac{\log_b(n)\cdot(\log_b(n)+1)}{2}\right) \\
&\in \Theta(nk\log(nk)\log(n)).
\end{aligned}$$

Chapter 9

Suffix Tree and the Longest Common Substring Problem

This chapter denotes strings of possibly many characters by using uppercase letters and denotes single characters by using lowercase letters.

9.1 Radix trie

A radix trie is a tree data structure that contains a collection of strings. The edges in a radix trie each have a single character label, denoted $label((u, v))$. Each node u in the tree has a unique path from the root (because it is a tree) and this path generates a string, which we denote $label(u)$. In this manner, each string in the collection can be recovered by the path from the root to some leaf.

Tries have a uniqueness property that ensures that no two paths will result in the same string. That is, if a node u in the tree has two outward edges (u, v_1) and (u, v_2), then the edge labels must have a different character: $label((u, v_1)) \neq label((u, v_2))$. For example, no valid trie would have two edges originating from u sharing the same prefix, such as $label((u, v_1)) =$ "a", $label((u, v_2)) =$ "a"; this would correctly be represented by "factoring out" the shared prefix and adding it on some path above, so that the edges to v_1 and v_2 are labeled with distinct characters. As a result, searching for a path that starts with some prefix should return a unique location (a node or a position along an edge) in the trie.

Radix tries can be inefficient because they store internal nodes even when

there is only a single path between them. For example, a radix trie of *abcd* will be a single path $\circ \underbrace{\rightarrow}_{a} \circ \underbrace{\rightarrow}_{b} \circ \underbrace{\rightarrow}_{c} \circ \underbrace{\rightarrow}_{d} \circ$.

9.2 Suffix tree

Given a string S, the suffix tree of that string is a compressed trie containing all suffixes $S_0 = S[0 \ldots n-1], S_1 = S[1 \ldots n-1], S_2 = S[2 \ldots n-1], \ldots S_{n-1} = S[n-1]$. The compressed property ensures that every internal node in the suffix tree has at least two children. Specifically, the compressed property states that an edge in the tree (u, v) between node u and node v will be contracted into a single node unless there is some other node w s.t. there is an edge (u, w)); thus, unlike a standard radix trie, which progresses with each edge representing an additional character, each edge may now represent a string of characters.

Because this data structure is a trie, then each path down from the root can be thought of as successively appending labels to a result string. To guarantee the uniqueness property when using multi-character edge labels, we now require that that all edges from node u must have labels starting with distinct characters.

9.2.1 Construction

A suffix tree can be constructed by inserting all suffixes in descending order of size; that is, by beginning with a single edge from the root to a leaf, with label S_1, and then inserting S_2, and then S_3, and so on until every suffix has been inserted.

This is, of course, trivial when every suffix begins with a thus far unseen character, as we can observe with the string $S = abcdefg\ldots$. In this case, the root of the tree has an edge to each suffix, because each new suffix inserted begins with a character that has never been the prefix of a label of an edge starting from the root, the encountered characters represent a split. The resulting suffix tree will have edges $label(root, S_1) =$ "$abcdefg\ldots$", $label(root, S_2) =$ "$bcdefg\ldots$", $label(root, S_3) =$ "$cdefg\ldots$", and so on; however, when a character is encountered twice, there will be a split. For example, the string $S = abac$ will insert the first two suffixes "abac" and "bac" trivially, because they both begin with new prefixes from the root. Inserting $S_3 =$ "ac" will find that a partial path from the root, along the edge $(root, S_1)$,

will begin with the same character "a". Thus, to ensure the trie property, we must search downard in the tree for S_i, the suffix we are inserting, and find the split point, where insertion will take place. This is equivalent to finding $head_i$, which is defined as the longest prefix shared with any already inserted suffix:

$$head_i = longest_{j<i} sharedPrefix(S_i, S_j),$$

where $sharedPrefix(S_i, S_j)$ is the longest prefix shared between the two suffix strings S_i and S_j. That is, $S_i[0 \dots k-1]$ where k is the maximum integer where $S_i[0 \dots k-1] = S_j[0 \dots k-1]$.

The location (which may be either a node or an edge) of $head_i$ in the trie is denoted $loc(head_i)$, where the path from the root to $loc(head_i)$ generates the string $head_i$. If $loc(head_i)$ lies on an edge (rather than on a node), then we can split the edge and insert an intermediate node so that $loc(head_i)$ lies along a node. Once we find $loc(head_i)$, we simply need to insert an edge from $loc(head_i)$ to a leaf with the label corresponding to the remainder of suffix S_i (denoted $tail_i$ where $S_i = head_i tail_i$). Note that we must create a new edge labeled $tail_i$ to a new leaf node, because we defined $head_i$ to be the maximal matching prefix, and thus any remaining characters (*i.e.* $tail_i$) must start with a character that doesn't match any path in the current suffix tree. If the string S is terminated with a "sentinel" character \$ that is not used in any other part of the string (*e.g.* $S = S[0]S[1]S[3] \dots \$$), then there will always be at least one remaining character after $head_i$ (*i.e.* $tail_i \neq$ "). This can be verified by observing that since every node u in the tree corresponds to a unique label $label(u)$, which has a unique "character depth", $|label(u)|$, and each of the suffixes inserted will have the sentinel character \$ occur exactly once at each character depth. Thus there will always be a branch to a new leaf node for every suffix inserted.

9.2.2 Achieving linear space construction

First, we show that we can achieve a linear number of nodes: As shown above, each insertion of a suffix S_i corresponds to always adding a new leaf node, and potentially inserts a new internal node. Thus, it is trivial to see that we insert at most $2n$ nodes (because each of the n suffixes corresponds to at most 2 nodes inserted).

Second, we can use the linear bound on the number of nodes to achieve

a linear space overall. At first, the possibility to achieve linear space may seem impossible: after all, the length of a particularly suffix $|S_i| = n - i + 1$, and thus the cumulative lengths of storing all suffix strings will be $\sum_i^n |S_i| = \sum_i^n n - i + 1 = n^2 + n - \frac{n(n+1)}{2}$, which is $\in \Theta(n^2)$; however, each of these strings is a contiguous block in the original string S, and thus, given a single copy of the full string S, we can simply store an integer for the ending point. Thus, for each suffix, we would simply store the ending point (which is a constant size).

Likewise, we use the same strategy to store the labels for the edges in the suffix tree; here we simply need to store the starting and ending points of the label in the string S. For example, in $S =$ "$abcghixyz$" the contiguous block "ghi" would be stored as the integer pair $(3, 5)$ to indicate that "ghi" $= S[3]S[4]S[5] = S[3 \ldots 5]$. In `Python`, these string "slices" will not copy the original string. Thus, each edge can be encoded using a constant size (two integers). We can also see that there are a linear number of edges (there are linearly many nodes, and each node of which has only one incoming edge, guaranteed by the fact that the graph is a tree). Thus, we can store the suffix tree in $O(n)$ space.

A simple suffix tree that uses $O(n)$ space is demonstrated in Listing 9.1. We call the standard method of searching for $head_i$ and inserting a suffix, wherein we scan character-by-character and move down the tree until we find a mismatch, "slowscan". We can see that the runtime of slowscan is linear in the number of characters that must be scanned; however, note that it may be possible to improve this by *starting* slowscan further down in the suffix tree.

Listing 9.1: A suffix tree that uses $O(n)$ space, but which is slow to construct.

```
def indent(depth):
  for i in xrange(depth):
    print ' ',

class Edge:
  # string should be a slice of a string (so that no copy is made)
  def __init__(self, string, source=None, dest=None):
    self.string = string
    self.source = source
    self.dest = dest

  def split(self, edge_offset):
    e_a = Edge(self.string[:edge_offset])
```

```
    e_b = Edge(self.string[edge_offset:])

    intermediate_node = Node(parent_edge=self,
        character_depth=self.source.character_depth+edge_offset)
    # self.source --> intermediate_node (via e_a)
    e_a.source = self.source
    e_a.dest = intermediate_node

    # intermediate_node --> self.dest (via e_b)
    e_b.source = intermediate_node
    e_b.dest = self.dest
    self.dest.parent_edge = e_b
    intermediate_node.set_edge_by_leading_character(e_b)
    e_b.dest.parent_edge = e_b

    # overwrite self so that anything that references this edge stays
        current:
    self.source = e_a.source
    self.dest = e_a.dest
    self.string = e_a.string

    return self,e_b

  def print_helper(self, depth):
    indent(depth)
    print '-->',self.string
    self.dest.print_helper(depth+1)

class Node:
  number_nodes=0
  def __init__(self, parent_edge=None, character_depth=None):
    self.parent_edge = parent_edge
    self.character_depth = character_depth
    self.first_char_to_edge = {}

    self.unique_id = Node.number_nodes
    Node.number_nodes += 1

  def parent(self):
    return self.parent_edge.source

  def set_edge_by_leading_character(self, edge):
    first_edge_char = edge.string[0]

    self.first_char_to_edge[first_edge_char] = edge
```

```
    edge.source = self

  def print_helper(self, depth):
    indent(depth)
    print '@',self.character_depth
    for e in self.first_char_to_edge.values():
      e.print_helper(depth+1)

  def is_root(self):
    return self.parent() == self

  def label(self):
    result = ""
    if not self.is_root():
      result = self.parent().label()
      result += self.parent_edge.string
    return result

class Location:
  def __init__(self, node, edge=None, edge_offset=None):
    self.node = node

    # either the edge and edge_offset are both None (the location is
    # at a node) or both are not None (the location is on an edge):
    assert( edge == None and edge_offset == None or edge != None and
        edge_offset != None)
    self.edge = edge
    self.edge_offset = edge_offset

  def on_node(self):
    return self.edge == None

  def on_edge(self):
    return not self.on_node()

  def create_node_here_if_necessary_and_return(self):
    if self.on_node():
      return self.node

    # on an edge; split the edge and add a new internal node
    e_a, e_b = self.edge.split(self.edge_offset)
    return e_a.dest

  def insert_suffix_here_and_return_node(self, suffix_tail):
    node = self.create_node_here_if_necessary_and_return()
```

```
    edge = Edge(suffix_tail)
    leaf = Node(parent_edge=edge, character_depth=node.character_depth +
        len(suffix_tail))
    edge.dest = leaf
    node.set_edge_by_leading_character(edge)
    return node

  def character_depth(self):
    if self.on_node():
      return self.node.character_depth
    return self.node.character_depth + self.edge_offset

  def label(self):
    result = self.node.label()
    if self.on_edge():
      result += self.edge.string[:self.edge_offset]
    return result

class SuffixTree:
  def __init__(self, string):
    self.string=string
    self.n = len(string)
    self.root=Node(character_depth=0)
    self.root.parent_edge = Edge('', self.root, self.root)

    self.build_tree()

  def build_tree(self):
    for i in xrange(self.n):
      suffix = self.string[i:]
      head_location = self.get_head(suffix)
      head_location.insert_suffix_here_and_return_node(suffix[head_location.character_depth():])

  # gets the deepest location in the tree that matches suffix:
  def get_head(self, suffix):
    return self.slow_scan(self.root, suffix)

  def slow_scan(self, start_node, suffix):
    if len(suffix) == 0:
      # return node location:
      return Location(start_node)

    first_char = suffix[0]
```

```
    if first_char not in start_node.first_char_to_edge:
      # return node location:
      return Location(start_node)

    # result must be on edge or on subtree after edge:
    edge = start_node.first_char_to_edge[first_char]

    # first character must match, so start at 1:
    for i in xrange( 1, min(len(suffix), len(edge.string)) ):
      suffix_char = suffix[i]
      edge_char = edge.string[i]

      if suffix_char != edge_char:
        # return edge location:
        return Location(start_node, edge, i)

    # a mismatch must exist before the end of suffix (because of
    # unique $ terminator character). thus, if loop terminates because
    # it goes past the end (instead of returning), then the match
    # extends past the end of the edge, and so the search should
    # continue at the destination node.
    return self.slow_scan(edge.dest, suffix[len(edge.string):])

  def print_helper(self):
    self.root.print_helper(0)

if __name__=='__main__':
  st = SuffixTree('ANAANY$')
  st.print_helper()
```

9.2.3 Achieving linear time construction

Clearly, finding $loc(head_i)$ is the only time-consuming task; once $loc(head_i)$ is found, insertion of suffix S_i takes $O(1)$ steps. The key to constructing the suffix tree in $O(n)$ is ensuring that the total time to find $loc(head_0), loc(head_1), \ldots loc(head_{n-1})$ is $\in O(n)$.

Suffix lemma If $|head_i| = k$, then there exists suffix j with $j < i$ such that $S[j \ldots j+k-1] = S[i \ldots i+k-1]$. By stripping off the first character, we see that $S[j+1 \ldots j+k-1] = S[i+1 \ldots i+k-1]$, and thus S_{i+1} matches at least the first $k-1$ characters of S_{j+1}; as a result, $|head_{i+1}| \geq k-1$, because there is some suffix $j+1$ matching at least the first $k-1$ characters.

To put it another way, if $head_i = xA$, then suffix $S_i = xA\ldots$ matches the prefix of some previous suffix $S_j = xA\ldots xA\ldots$ (where $j < i$). Then, when searching for $head_{i+1}$, we notice that $S_{i+1} = A\ldots$ (*i.e.*, the subsequent suffix, which can be found by removing the first character x from the previous suffix S_i), must also match the previous suffix $S_{j+1} = A\ldots xA\ldots$. If $|head_i| = k$, we are guaranteed that the first $k-1$ characters of S_{i+1} match S_{j+1}, and thus $|head_{i+1}| \geq k-1$ (we are not guaranteed strict equality, because it is possible that *another* suffix $S_{j'}$ matches even better for some $j' < i$).

For example, let $S = xyAaAbyAb\$$:

$$\begin{aligned} S_1 &= xyAaAbyAb\$ \\ S_2 &= yAaAbyAb\$ \\ S_3 &= AaAbyAb\$ \\ &\vdots \\ S_{j'} &= AbyAb\$ \\ &\vdots \\ S_i &= yAb\$ \\ S_{i+1} &= Ab\$ \end{aligned}$$

S_i matches $S_j = S_2$ with shared prefix yA. The suffix lemma implies that S_{i+1} must at least match $S_{j+1} = S_3$ with shared prefix A; however, there may be a better match: consider that S_{i+1} matches $S_{j'}$ with shared prefix Ab, which is one character longer than the shared prefix with S_{j+1}. Thus $|head_{i+1}| \geq |head_i| - 1$, because we are guaranteed the match to S_{j+1}, but a better match may exist.

"Fastscan" We can exploit the suffix lemma to show that if $|head_i| = k$, when we insert the subsequent suffix S_{i+1}, we are already guaranteed that the first $k-1$ characters of S_{i+1} are already found in the current state of the suffix tree (*i.e*, that prefix is already found in the suffix tree after inserting the first i suffixes $S_1, S_2, \ldots, S_i$).

Since we know that these first $k-1$ characters are already in the tree, we simply look at the first *character* each time we take an edge, in order to find which edge to take. Then, if the first character of the edge matches the next character in the $k-1$ length substring, and if the edge contains m characters, we simply follow the edge and then arrive at the next node needing to search

for the remaining $k-1-m$ characters, and having jumped m characters forward in our target substring. Note that if m is ever longer than the number of characters remaining, the target location is in the middle of the edge. Thus, the runtime of this insertion, which we call "fastscan" is linear, but rather than being linear in the number of characters (like the slowscan search algorithm), it is linear in the number of *edges* processed. Again, as in the slowscan algorithm, we note that this search could be performed more efficiently if we are able to start the search further down in the tree.

Runtime of slowscan When using the suffix lemma and fastscan whenever possible, the runtime of slowscan improves dramatically. If $|head_{i-1}| = k$, then the suffix lemma states that $|head_i| \geq k-1$, and thus fastscan will descend the tree $k-1$ characters. Thus, if we use the location where fastscan terminates as the start of our slowscan search, the number of characters descended by slowscan must be

$$\begin{cases} |head_i| - (|head_{i-1}| - 1) & |head_{i-1}| \geq 1 \\ |head_i| & \text{else.} \end{cases}$$

When $|head_{i-1}| = 0$, then $|head_i| - (|head_{i-1}| - 1) = |head_i| - |head_{i-1}| + 1 = |head_i| + 1$ overestimates the cost by 1; however, that 1 extra step can be amortized out because the overhead of calling slowscan must be $\in O(1)$; therefore, we will always consider the runtime per slowscan call to be $|head_i| - (|head_{i-1}| - 1)$

Hence, the total cost of slowscan is found by a telescoping sum over the slowscan costs in each iteration:

$$\begin{aligned} \sum_i^n |head_i| - (|head_{i-1}| - 1) = \quad & |head_n| - |head_{n-1}| + 1 + \\ & |head_{n-1}| - |head_{n-2}| + 1 + \\ & \ldots + \\ & |head_2| - |head_1| + 1 \\ = \quad & n - 1 + |head_n| - |head_1| \\ = \quad & n - 1 + 0 - 0 \\ \in \quad & \Theta(n). \end{aligned}$$

We know that $|head_1| = 0$, because when S_1 is inserted, no other suffixes have yet been inserted, and thus there is no chance for a prefix to overlap with

any previous suffix. We know that $|head_n| = 0$ because $|head_n|$ comes from S_n =\$, and no other suffix is allowed to begin with \$, and thus $|head_n| = 0$. Listing 9.2 shows an implementation that uses the suffix lemma in this manner.

Listing 9.2: A suffix tree that exploits the suffix lemma by using fastscan. The worst-case runtime is still not linear.

```
def indent(depth):
  for i in xrange(depth):
    print ' ',

class Edge:
  # string should be a slice of a string (so that no copy is made)
  def __init__(self, string, source=None, dest=None):
    self.string = string
    self.source = source
    self.dest = dest

  def split(self, edge_offset):
    e_a = Edge(self.string[:edge_offset])
    e_b = Edge(self.string[edge_offset:])

    intermediate_node = Node(parent_edge=self,
        character_depth=self.source.character_depth+edge_offset)
    # self.source --> intermediate_node (via e_a)
    e_a.source = self.source
    e_a.dest = intermediate_node

    # intermediate_node --> self.dest (via e_b)
    e_b.source = intermediate_node
    e_b.dest = self.dest
    self.dest.parent_edge = e_b
    intermediate_node.set_edge_by_leading_character(e_b)
    e_b.dest.parent_edge = e_b

    # overwrite self so that anything that references this edge stays
        current:
    self.source = e_a.source
    self.dest = e_a.dest
    self.string = e_a.string

    return self,e_b

  def print_helper(self, depth):
```

```
    indent(depth)
    print '-->',self.string
    self.dest.print_helper(depth+1)

class Node:
  number_nodes=0
  def __init__(self, parent_edge=None, character_depth=None):
    self.parent_edge = parent_edge
    self.character_depth = character_depth
    self.first_char_to_edge = {}

    self.unique_id = Node.number_nodes
    Node.number_nodes += 1

  def parent(self):
    return self.parent_edge.source

  def set_edge_by_leading_character(self, edge):
    first_edge_char = edge.string[0]

    self.first_char_to_edge[first_edge_char] = edge
    edge.source = self

  def print_helper(self, depth):
    indent(depth)
    print '@',self.character_depth
    for e in self.first_char_to_edge.values():
      e.print_helper(depth+1)

  def is_root(self):
    return self.parent() == self

  def label(self):
    result = ""
    if not self.is_root():
      result = self.parent().label()
      result += self.parent_edge.string
    return result

class Location:
  def __init__(self, node, edge=None, edge_offset=None):
    self.node = node

    # either the edge and edge_offset are both None (the location is
    # at a node) or both are not None (the location is on an edge):
```

```
    assert( edge == None and edge_offset == None or edge != None and
        edge_offset != None)
    self.edge = edge
    self.edge_offset = edge_offset

  def on_node(self):
    return self.edge == None

  def on_edge(self):
    return not self.on_node()

  def create_node_here_if_necessary_and_return(self):
    if self.on_node():
      return self.node

    # on an edge; split the edge and add a new internal node
    e_a, e_b = self.edge.split(self.edge_offset)
    return e_a.dest

  def insert_suffix_here_and_return_node(self, suffix_tail):
    node = self.create_node_here_if_necessary_and_return()

    edge = Edge(suffix_tail)
    leaf = Node(parent_edge=edge, character_depth=node.character_depth +
        len(suffix_tail))
    edge.dest = leaf
    node.set_edge_by_leading_character(edge)
    return node

  def character_depth(self):
    if self.on_node():
      return self.node.character_depth
    return self.node.character_depth + self.edge_offset

  def label(self):
    result = self.node.label()
    if self.on_edge():
      result += self.edge.string[:self.edge_offset]
    return result

class SuffixTree:
  def __init__(self, string):
    self.string=string
    self.n = len(string)
    self.root=Node(character_depth=0)
```

```
    self.root.parent_edge = Edge('', self.root, self.root)

    self.index_to_head_node = {}

    self.build_tree()

  def build_tree(self):
    for suffix_i in xrange(self.n):
      self.insert_suffix(suffix_i)

  # gets the deepest location in the tree that matches suffix:
  def insert_suffix(self, suffix_i):
    suffix = self.string[suffix_i:]

    if suffix_i>0 and not self.index_to_head_node[suffix_i-1].is_root():
      prev_head = self.index_to_head_node[suffix_i-1]
      characters_that_must_match = prev_head.character_depth-1

      start_node = self.root
      remaining_characters_that_must_match = characters_that_must_match

      loc = self.fast_scan(start_node, suffix,
          remaining_characters_that_must_match)
      loc = self.slow_scan(loc, suffix)
    else:
      loc = self.slow_scan(Location(self.root), suffix)

    head_i =
        loc.insert_suffix_here_and_return_node(suffix[loc.character_depth():])
    self.index_to_head_node[suffix_i] = head_i

  def fast_scan(self, start_node, suffix, characters_that_must_match):
    sub_suffix = suffix[start_node.character_depth:]
    if characters_that_must_match <= 0:
      return Location(start_node)

    first_char = sub_suffix[0]

    edge = start_node.first_char_to_edge[first_char]

    if len(edge.string) <= characters_that_must_match:
      # cross entire edge:
      return self.fast_scan(edge.dest, suffix,
          characters_that_must_match-len(edge.string))
    else:
```

```
    # finishes along edge:
    edge_offset = characters_that_must_match
    return Location(start_node, edge, edge_offset)

def slow_scan(self, start_loc, suffix):
  sub_suffix = suffix[start_loc.character_depth():]

  if len(suffix) == 0:
    # return node location:
    return start_loc

  first_char = suffix[start_loc.character_depth()]

  # if node, try looking up correct edge as below
  # if edge location, simply extend

  # If start_loc is a node location:
  if start_loc.on_node():
    if first_char not in start_loc.node.first_char_to_edge:
      # return node location:
      return start_loc

    # result must be on edge or on subtree after edge:
    edge = start_loc.node.first_char_to_edge[first_char]
    start_index_on_edge = 0
  else:
    # start_loc is an edge location:
    edge = start_loc.edge
    start_index_on_edge = start_loc.edge_offset

  # edge is now known in either case:
  for i in xrange(0, min(len(sub_suffix),
      len(edge.string)-start_index_on_edge) ):
    suffix_char = sub_suffix[i]
    edge_char = edge.string[start_index_on_edge+i]

    if suffix_char != edge_char:
      # return edge location:
      return Location(start_loc.node, edge, start_index_on_edge+i)

  # a mismatch must exist before the end of sub_suffix (because of
  # unique $ terminator character). thus, if loop terminates because
  # it goes past the end (instead of returning), then the match
  # extends past the end of the edge, and so the search should
  # continue at the destination node.
```

```
    return self.slow_scan(Location(edge.dest), suffix)

  def print_helper(self):
    self.root.print_helper(0)

if __name__=='__main__':
  st = SuffixTree('ANAANY$')
  st.print_helper()
```

Fastscan is not yet sufficient to guarantee linear time construction. The reason for this is that it is possible to choose a string such that the final suffix tree will have several nodes with high "node depth" (*i.e.*, the standard depth in the tree, counted by the number of edges traveled to reach a given point).

Suffix links After inserting suffix S_i, we have computed already computed (and added a node for) $loc(head_i)$. In the previous iteration, we have done the same for $loc(head_{i-1})$; thus, if we store those previous nodes $loc(head_j)$ in an array, for all $j < i$, we can easily add a "suffix link" from $link(loc(head_{i-1})) = loc(head_i)$. Each of these suffix links are special edges that connect the node with label xA to the node with label A.

Unfortunately, we would like to use that link *in* iteration i in order to find $loc(head_i)$, but we *create* the suffix link in iteration i *after* finding $loc(head_i)$; so thus far, the cache does not add any help; however, the parent of $loc(head_{i-1})$ has a suffix link that was created when it was inserted (it was inserted in a previous iteration), and thus we can follow the suffix link $link(parent(loc(head_{i-1})))$, which will not go directly to $loc(head_i)$, but will reach an ancestor of $loc(head_i)$, partially completing the search for $loc(head_i)$. For this reason, whenever there is any benefit to the suffix link (*i.e.*, whenever $parent(loc(head_{i-1})) \neq root$), fastscan can begin at $link(parent(loc(head_{i-1})))$.

The node at $link(parent(loc(head_{i-1})))$ is guaranteed to be an ancestor of $loc(head_i)$ (or, equivalently, $link(loc(head_{i-1}))$); this is true because following a link is equivalent to removing the first character on the left, while following a parent edge is equivalent to removing some nonzero number of characters from the right: $link(loc(xA \ldots B)) = loc(A \ldots B)$ and $link(parent(loc(xA \ldots B))) = loc(A)$, and because A is a prefix of $A \ldots B$, $loc(A)$ is an ancestor of $loc(A \ldots B)$.

As before, fastscan will find the $|head_{i-1}| - 1$ length prefix that we

know must already be in the tree, but where we have already matched $|label(loc(parent(head_{i-1})))|$ characters by taking the suffix link. And then, after completing fastscan, we perform slowscan to find any additional matching characters.

A suffix-link based implementation is shown in Listing 9.3. We will prove that that implementation is constructed in $O(n)$.

Listing 9.3: A suffix tree that uses suffix links to achieve $O(n)$ construction.

```
def indent(depth):
  for i in xrange(depth):
    print ' ',

class Edge:
  # string should be a slice of a string (so that no copy is made)
  def __init__(self, string, source=None, dest=None):
    self.string = string
    self.source = source
    self.dest = dest

  def split(self, edge_offset):
    e_a = Edge(self.string[:edge_offset])
    e_b = Edge(self.string[edge_offset:])

    intermediate_node = Node(parent_edge=self,
        character_depth=self.source.character_depth+edge_offset)
    # self.source --> intermediate_node (via e_a)
    e_a.source = self.source
    e_a.dest = intermediate_node

    # intermediate_node --> self.dest (via e_b)
    e_b.source = intermediate_node
    e_b.dest = self.dest
    self.dest.parent_edge = e_b
    intermediate_node.set_edge_by_leading_character(e_b)
    e_b.dest.parent_edge = e_b

    # overwrite self so that anything that references this edge stays
        current:
    self.source = e_a.source
    self.dest = e_a.dest
    self.string = e_a.string

    return self,e_b
```

```
  def print_helper(self, depth):
    indent(depth)
    print '-->',self.string
    self.dest.print_helper(depth+1)

class Node:
  number_nodes=0
  def __init__(self, parent_edge=None, link=None, character_depth=None):
    self.parent_edge = parent_edge
    self.character_depth = character_depth
    self.first_char_to_edge = {}
    self.link = link

    self.unique_id = Node.number_nodes
    Node.number_nodes += 1

  def parent(self):
    return self.parent_edge.source

  def set_edge_by_leading_character(self, edge):
    first_edge_char = edge.string[0]

    self.first_char_to_edge[first_edge_char] = edge
    edge.source = self

  def print_helper(self, depth):
    indent(depth)
    print '@',self.character_depth,
    if self.link != None:
      print 'Link',self.link.unique_id, "'"+self.label()+"'", 'to',
          "'"+self.link.label()+"'"
    else:
      print
    for e in self.first_char_to_edge.values():
      e.print_helper(depth+1)

  def is_root(self):
    return self.parent() == self

  def label(self):
    result = ""
    if not self.is_root():
      result = self.parent().label()
      result += self.parent_edge.string
    return result
```

```
class Location:
  def __init__(self, node, edge=None, edge_offset=None):
    self.node = node

    # either the edge and edge_offset are both None (the location is
    # at a node) or both are not None (the location is on an edge):
    assert( edge == None and edge_offset == None or edge != None and
        edge_offset != None)
    self.edge = edge
    self.edge_offset = edge_offset

  def on_node(self):
    return self.edge == None

  def on_edge(self):
    return not self.on_node()

  def create_node_here_if_necessary_and_return(self):
    if self.on_node():
      return self.node

    # on an edge; split the edge and add a new internal node
    e_a, e_b = self.edge.split(self.edge_offset)
    return e_a.dest

  def insert_suffix_here_and_return_node(self, suffix_tail):
    node = self.create_node_here_if_necessary_and_return()

    edge = Edge(suffix_tail)
    leaf = Node(parent_edge=edge, character_depth=node.character_depth +
        len(suffix_tail))
    edge.dest = leaf
    node.set_edge_by_leading_character(edge)
    return node

  def character_depth(self):
    if self.on_node():
      return self.node.character_depth
    return self.node.character_depth + self.edge_offset

  def label(self):
    result = self.node.label()
    if self.on_edge():
      result += self.edge.string[:self.edge_offset]
```

```
    return result

class SuffixTree:
  def __init__(self, string):
    self.string=string
    self.n = len(string)
    self.root=Node(character_depth=0)
    self.root.parent_edge = Edge('', self.root, self.root)

    self.index_to_head_node = {}

    self.build_tree()

  def build_tree(self):
    for suffix_i in xrange(self.n):
      #print 'adding ', self.string[suffix_i:]
      self.insert_suffix(suffix_i)

  # gets the deepest location in the tree that matches suffix:
  def insert_suffix(self, suffix_i):
    suffix = self.string[suffix_i:]

    if suffix_i>0 and not self.index_to_head_node[suffix_i-1].is_root():
      # using suffix lemma
      prev_head = self.index_to_head_node[suffix_i-1]
      characters_that_must_match = prev_head.character_depth-1

      if prev_head.parent() == self.root:
        # no suffix link available:
        start_node = self.root
        remaining_characters_that_must_match = characters_that_must_match
      else:
        # suffix link available:
        start_node = prev_head.parent().link
        remaining_characters_that_must_match = characters_that_must_match
            - start_node.character_depth

      loc = self.fast_scan(start_node, suffix,
          remaining_characters_that_must_match)

      # if xA is not found at root, a link is created. thus, every
      # internal node must have a suffix link.

      # add suffix link from xA to A (A would be found by fast_scan):
      link_dest_node = loc.create_node_here_if_necessary_and_return()
```

```
      self.index_to_head_node[suffix_i-1].link = link_dest_node
      loc = self.slow_scan(loc, suffix)
    else:
      # not using suffix lemma
      loc = self.slow_scan(Location(self.root), suffix)

    head_i =
        loc.insert_suffix_here_and_return_node(suffix[loc.character_depth():])
    self.index_to_head_node[suffix_i] = head_i

  def fast_scan(self, start_node, suffix, characters_that_must_match):
    sub_suffix = suffix[start_node.character_depth:]
    if characters_that_must_match <= 0:
      return Location(start_node)

    first_char = sub_suffix[0]

    edge = start_node.first_char_to_edge[first_char]

    if len(edge.string) <= characters_that_must_match:
      # cross entire edge:
      return self.fast_scan(edge.dest, suffix,
          characters_that_must_match-len(edge.string))
    else:
      # finishes along edge:
      edge_offset = characters_that_must_match
      return Location(start_node, edge, edge_offset)

  def slow_scan(self, start_loc, suffix):
    sub_suffix = suffix[start_loc.character_depth():]

    if len(suffix) == 0:
      # return node location:
      return start_loc

    first_char = suffix[start_loc.character_depth()]

    # if node, try looking up correct edge as below
    # if edge location, simply extend

    # If start_loc is a node location:
    if start_loc.on_node():
      if first_char not in start_loc.node.first_char_to_edge:
        # return node location:
        return start_loc
```

```
      # result must be on edge or on subtree after edge:
      edge = start_loc.node.first_char_to_edge[first_char]
      start_index_on_edge = 0
    else:
      # start_loc is an edge location:
      edge = start_loc.edge
      start_index_on_edge = start_loc.edge_offset

    # edge is now known in either case:
    for i in xrange(0, min(len(sub_suffix),
        len(edge.string)-start_index_on_edge) ):
      suffix_char = sub_suffix[i]
      edge_char = edge.string[start_index_on_edge+i]

      if suffix_char != edge_char:
        # return edge location:
        return Location(start_loc.node, edge, start_index_on_edge+i)

    # a mismatch must exist before the end of sub_suffix (because of
    # unique $ terminator character). thus, if loop terminates because
    # it goes past the end (instead of returning), then the match
    # extends past the end of the edge, and so the search should
    # continue at the destination node.
    return self.slow_scan(Location(edge.dest), suffix)

  def print_helper(self):
    self.root.print_helper(0)

if __name__=='__main__':
  st = SuffixTree('ANAANY$')
  st.print_helper()
```

Every internal node will get a suffix link Except for the root, every internal (*i.e.*, every non-leaf) node will be assigned a link.

We only add internal nodes in two places in the code: The first place in the code is when we create a node at $loc(head_i)$ (as long as $loc(head_i) \neq root$) when S_i is inserted into the tree. If $loc(head_i) \neq root$, then we are assured that $loc(head_i)$ will be assigned a link in the following iteration. The second place in the code where we add an internal node is when we are making a suffix link from $loc(head_{i-1})$ to the location of where fastscan terminates in iteration i; in that case, we may add an internal node for the destination of

this suffix link if the fastscan result for S_i terminates along an edge. That node will be created only if no node already exists at the location where fastscan terminates.

In the second case above, it is not yet clear whether we are assured that subsequent iterations will add a link from that new internal node. Because iteration $i-1$ adds a node at $loc(head_{i-1})$ (if $loc(head_{i-1}) \neq root$), we know that the character of S_{i-1} immediately following $head_{i-1}$ distinguishes S_{i-1} from all S_j for $j < i$. Let's write $S_{i-1} = xAu\ldots$ and $head_{i-1} = xAu$, where all previous suffixes S_j beginning with xA have some other character following: $xAv\ldots$. The following iteration must likewise be the first time Au was split from Av (which must already exist in the tree) because they would be inserted in the iteration immediately after one of the already present $S_j = xAv\ldots$ was inserted.

In that case, $head_i = A$. Likewise, fastscan at iteration i will terminate at $loc(A)$. Thus, we see that $loc(head_i)$ (where an internal node is added using the first case above) will be at the same location as the second case listed above.

Since we've already shown that the first case above, we achieve the invariant that every internal node will be assigned a link from itself in the iteration after it was created.

Note that if $parent(loc(head_{i-1})) \neq root$, then $link(parent(loc(head_{i-1})))$ must exist during iteration i. In iteration i, there is only one node that may not have a suffix link coming from it: $loc(head_{i-1})$, which will be assigned a link from it at the end of iteration i. In a tree, a node cannot be its own parent, and therefore, if $parent(loc(head_{i-1})) \neq root$, then $link(parent(loc(head_{i-1})))$ must exist during iteration i.

Bounding node depth change from following suffix links Taking the parent node trivially decreases the node depth by 1. Taking a suffix link is more complicated:

An internal node will exist if and only if two suffixes that have the same prefix and both suffixes have a different character after that internal node. Given a string $S = \ldots xyAv\ldots$, where there is a suffix link from $S_j = xyAv\ldots$ to $S_i = yAv\ldots$ (where $j < i$), an internal node will only exist if there exists another suffix $S_{j'} = xyAw\ldots$, so that the after the shared prefix xyA, one has an edge starting with v and the other has an edge starting with w. The same will be true for the shorter strings: $S_{i'} = yAw\ldots$ must also

create a split with $S_i = yAv\ldots$, because both share a common prefix yA, but where (like their longer counterparts S_j and $S_{j'}$), there is a split from this internal node where one takes an edge starting v and the other takes an edge starting w. Thus, when suffixes are linked, every split (and hence every internal node added) to the longer suffix will result in an internal node added to a shorter suffix. There is one exception: the first character x found in S_j and $S_{j'}$ may also add an additional internal node because of a potential split with some *other* suffix $S_{j''} = xz\ldots$. Thus, with the exception of the single possible split utilizing the first character of both S_j and $S_{j'}$, every internal node inserted above $loc(head_j)$ will be inserted above $loc(head_i)$. Thus the maximum decrease in depth from following a suffix link is 1 node, because it is possible that there is at most 1 ancestor in $loc(head_j)$ that does not correspond to an ancestor in $loc(head_i)$.

Thus, we see that following the link of the parent will decrease the depth by no more than 2.

Runtime of fastscan The maximum depth of the tree is n. Because the node depth decrease from following the suffix link of a parent node will be at most 2 nodes, we will make an amortized argument that the total node depth increase of fastscan calls (which is equivalent to the total runtime of fastscan calls) must be in $O(n)$.

During iteration i, the depth after following a suffix link and performing fastscan is equal to the starting node depth (after following the suffix link) plus the runtime from using fastscan. This new depth will be $depthAfterFastscan_i \geq depth(loc(head_{i-1})) - 2 + fastscanCost_i$. We also know that $depth(loc(head_i)) \geq depthAfterFastscan_i$, because the slowscan operations made after finding the result of fastscan will only increase the depth. Thus, $depth(loc(head_i)) \geq depth(loc(head_{i-1})) - 2 + fastscanCost_i$, which can be rewritten to say that $fastscanCost_i \leq depth(loc(head_i)) - depth(loc(head_{i-1})) + 2$. The sum of the fastscan costs can thus be bounded by a telescoping sum:

$$\sum_i^n fastscanCost_i \leq$$
$$depth(loc(head_n)) - depth(loc(head_{n-1})) + 2 +$$
$$depth(loc(head_{n-1})) - depth(loc(head_{n-2})) + 2 + \cdots +$$
$$depth(loc(head_2)) - depth(loc(head_1)) + 2,$$

which collapses to $depth(loc(head_n)) - depth(loc(head_1)) + 2n$. Furthermore, because the depth of a suffix tree is at most n, the depth increase from $depth(loc(head_n)) - depth(loc(head_1)) \leq n$, and thus $\sum_i^n fastscanCost_i \leq 3n$.

Thus the total runtime necessary to find $loc(head_i)$ in every iteration is $O(n)$ (*i.e.*, the amortized runtime to find it in each iteration is constant), and thus an edge to a new leaf node can be added trivially to result in an $O(n)$ overall construction time.

9.3 Linear time/space solution to LCS

The longest common substring (LCS) problem finds the largest contiguous string common in two string arguments, A and B. Amazingly, the longest common substring problem can be solved by a suffix tree in $O(n)$ time and space.

We begin by concatenating the strings with unique terminator characters # and \$ to form a single string $S = A\#B\$$. Then we build a suffix tree of S. Any substring X found in both A and B must begin at least two suffixes of the string S: one beginning before the # (*i.e.*, $S_j = A \ldots \# \ldots \$$) and one found after the # (*i.e.*, $S_i = A \ldots \$$, where $j < i$). Thus, the suffixes S_i and S_j share a common prefix, A. Thus, the location of the longest common substring of A and B must have at least one descendent containing the # terminator (meaning the string occurs in A) and at least one descendent *not* containing the # character (meaning the string occurs in B).

Proof that the solution occurs on a node Furthermore, if A is the maximal matching prefix of S_j and S_i with $|C| = k$, then the character $S_i[k+1] \neq S_j[k+1]$ (or else adding one more character would produce a better substring, contradictig the notion that C is the maximal matching prefix). When this differentiating character is encountered during construction, it must result in a split in the tree, and the internal node inserted for that split will have label C. We can easily determine whether a node can be found in both strings because it will have at least one descendent containing # and at least one descendent not containing #.

Proof that solutions do not cross the # boundary Because we have concatenated the two strings into S, we must be careful that any label cross-

ing the boundary (*i.e.* $C = D \ldots \# \ldots E\$$) will not be recognized as a potential solution. This is trivial, because the node corresponding to such a string *cannot* have a descendent leaf node that does not contain #.

Finding the longest common substring Simple dynamic programming can be used to mark the nodes that have at least one descendent containing # and at least one descendent not containing #. Then, of the nodes satisfying both criteria (and thus with labels corresponding to strings found in both A and B) are ranked by their character depth. The label of the satisfying node with maximum character depth is the longest common substring found in both A and B. This is shown in Listing 9.4.

Listing 9.4: An $O(n)$ solution to the longest common substring problem.

```
from suffix_tree_links import *

def mark_if_descendant_has_first_terminator(node):
  for e in node.first_char_to_edge.values():
    child = e.dest
    mark_if_descendant_has_first_terminator(child)

  if len(node.first_char_to_edge) != 0:
    # not leaf
    res = False
    for e in node.first_char_to_edge.values():
      child = e.dest
      res = res or child.descendant_has_first_terminator
    node.descendant_has_first_terminator = res
  else:
    # leaf

    # terminators can only happen once in any string, so it should be on
        an edge into a leaf
    edge_into_leaf = node.parent_edge
    node.descendant_has_first_terminator = '#' in edge_into_leaf.string

def mark_if_descendant_does_not_have_first_terminator(node):
  for e in node.first_char_to_edge.values():
    child = e.dest
    mark_if_descendant_does_not_have_first_terminator(child)

  if len(node.first_char_to_edge) != 0:
    # not leaf
    res = False
```

```
    for e in node.first_char_to_edge.values():
      child = e.dest
      res = res or child.descendant_does_not_have_first_terminator
    node.descendant_does_not_have_first_terminator = res
  else:
    # leaf

    # terminators can only happen once in any string, so it should be on
        an edge into a leaf
    edge_into_leaf = node.parent_edge
    node.descendant_does_not_have_first_terminator = not '#' in
        edge_into_leaf.string

def all_descendants(node):
  result = [node]

  for e in node.first_char_to_edge.values():
    child = e.dest
    result.extend( all_descendants(child) )

  return result

def lcs(suffix_tree):
  mark_if_descendant_has_first_terminator(suffix_tree.root)
  mark_if_descendant_does_not_have_first_terminator(suffix_tree.root)

  all_nodes = all_descendants(suffix_tree.root)
  candidates = [ n for n in all_nodes if
      n.descendant_has_first_terminator and
      n.descendant_does_not_have_first_terminator ]
  best_character_depth = max([ node.character_depth for node in
      candidates ])
  best_nodes = [ node for node in candidates if node.character_depth ==
      best_character_depth ]

  # select an arbitrary solution in the case of a tie:
  return best_nodes[0].label()

if __name__ == '__main__':
#  X = 'BANAANABANAANAABANANAABANABANANABANANABANANNNA'
#  Y = 'ANBANABANABANANANNNNABANANABANANAAABANAAAAAABB'
  X = 'BANANA'
  Y = 'ANANAS'

  concatenated = X+'#'+Y+'$'
```

```
st = SuffixTree(concatenated)

print lcs(st)
```

These dynamic programming steps take $O(n)$ total, and thus the runtime of finding the longest common substring is, remarkably, $\in O(n)$. Since the cost of even reading the data is $\in \Omega(n)$, the longest common substring problem is solved $\in \Theta(n)$.

9.4 Practice and discussion questions

1. Consider $S = oompaloompa\$$.

   (a) Write all suffixes, $S_1, S_2, \ldots S_n$ for $S = oompaloompa\$$.

   (b) What is $head_6$?

   (c) What is $head_7$?

   (d) What does the suffix lemma guarantee about $head_8$?

   (e) What string would we use for fastscan when inserting S_8?

   (f) How many character would we compare with slowscan when inserting S_8 (after fastscan has already been run)?

2. Consider $S = abcdzabzabcd\$$.

   (a) What is S_8?

   (b) What is $head_8$?

   (c) What does the suffix lemma guarantee about $head_9$?

   (d) What is $head_9$?

9.5 Answer key

1. (a)

$$\begin{aligned} S_1 &= oompa\ell oompa\$ \\ S_2 &= ompa\ell oompa\$ \\ S_3 &= mpa\ell oompa\$ \\ S_4 &= pa\ell oompa\$ \\ S_5 &= a\ell oompa\$ \\ S_6 &= \ell oompa\$ \\ S_7 &= oompa\$ \\ S_8 &= ompa\$ \\ S_9 &= mpa\$ \\ S_{10} &= pa\$ \\ S_{11} &= a\$ \\ S_{12} &= \$ \end{aligned}$$

   (b) S_6 is the first suffix to start with ℓ; therefore, $|head_6| = 0$ and $head_6 =$ "", the empty string.

   (c) S_7 has its largest shared prefix with S_1: this prefix is *oompa*, and therefore $head_7 = oompa$.

   (d) The suffix lemma guarantees that $|head_i| \geq |head_{i-1}| - 1$; therefore, $|head_8| \geq |oompa| - 1 = 4$.

   (e) By using the first 4 characters (4 because of the answer to the question above) of S_8, we see that we could call fastscan knowing that *ompa* is already in the tree. This would thus be the query we used for the fastscan call.

   (f) Slowscan would need to compare the characters following *ompa* in S_8 until the first mismatch in the tree. There was only one suffix before S_8 that has prefix *ompa*: $S_2 = ompa\ell oompa\$$. The first mismatch will occur between the character \$ in S_8 and the character ℓ in S_2. This was the first character compared with slowscan, so we will only compare 1 character with slowscan after fastscan has run.

2. (a) $S_8 = zabcd\$$
   (b) $head_8 = zab$ by matching $S_5 = zabzabcd\$$.
   (c) $|head_9| \geq |head_8| - 1 = 2$
   (d) $S_9 = abcd\$$ and $head_9 = abcd$ by matching S_1. Note that this is an example where the suffix lemma does not reach equality (*i.e.*, the suffix lemma gives only a prefix of $head_9$, not the full value of $head_9$).

Chapter 10

Minimum Spanning Tree and Approximation of the Traveling Salesman Problem

10.1 Minimum spanning tree

A tree T is defined as a graph without any cycles[1]. If edges are weighted, then we can define the minimum spanning tree (MST) of a graph G. The MST is the tree with smallest total edge weight that connects every node in G. Note that G must be a connected graph for any spanning tree, let alone a minimum spanning tree, to exist. We denote this MST as T^*, which will be a subgraph of G (*i.e.*, the edges of the graph T^* must be a subset of the edges of G).

Finding the MST of a weighted graph feels like a combinatorial problem, the kind of thing that could easily be much more difficult than it would seem; however, we can use properties of the MST to find it easily using either of two different approaches.

The key properties of MST that we will employ are as follows:

1. **Cut property:** Any two subtrees of T^* will be connected through the minimum weight edge in G that connects the nodes in those subtrees.

2. **Cycle property:** Any cycle in G cannot be present in T^*. An edge with higher weight than any other edge in the cycle must be excluded

[1]Equivalently, a tree is a graph where there is a unique path between any pair of nodes.

from T^*.

Both of these properties can be proven by contradiction:

For the cut property, denote e as the edge that connects the two disjoint subtrees. Assume the contrary, that another edge e' with weight lower then e in the graph with lower weight connects the two disjoint subtrees. In that case, then exchaning e for e' will lower the total cost of the edge weights in the tree, contradicting the definition of T^*. This contradiction implies the cut property. Note that when multiple edges connect the two disjoint subtrees and achieve the minimum weight, then either can be used.

For the cyle property, assume the contrary, that the maximum weight edge in the cycle, $e = (a, b)$, belongs to T^*. Removing e must break T^* into two disjoint subtrees with node sets V_1 and V_2 with $a \in V_1$ and $b \in V_2$. A cycle is defined as having exactly two distinct paths between every pair of nodes in the cycle; therefore, because the cycle connects V_1 to V_2, the cycle contains another distinct path p (*i.e.*, the path p does not include e) between a and b. Path p must contain an edge $e' \neq e$ crossing from V_1 to V_2; that edge reconnects V_1 to V_2 and does so with weight less than the maximum weight edge. Thus exchanging e for e' achieves a spanning tree with lower total weight, which contradicts the definition of T^*. This contradiction implies the cycle property. As with the cut property, when two edges in the cycle bridge V_1 and V_2 with equal weight, then an arbitrary edge can be used.

10.1.1 Prim's algorithm

Prim's algorithm is based on the cut property. Start by including an arbitrary node a in T^* (the identity of this node does not matter since every node must eventually be included in a spanning tree). The "cut" between $\{a\}$ and all other nodes $V \setminus \{a\}$ is the set of edges that connect $\{a\}$ to $V \setminus \{a\}$. The edge in the cut with minimum weight is added to T^*. The process is repeated by cutting between the vertices currently in T^* and the vertices not yet in T^*. In the case where two or more edges in a cut achieve the minimum weight in the cut, an arbitrary edge of those achieving the minimum edge weight can be used. This is shown in Listing 10.1.

The proof of correctness follows directly from the cut property. In each stage, two sets of vertices are considered, and an edge in the cut between them with minimum weight is used to join them. Thus by selecting these minimum weight edges, we achieve an MST.

Prim's algorithm can be implemented in a straightforward naive manner. In each iteration i from 1 to n (where n is the number of nodes in the graph), the cut will be of size $i \times (n-i)$. This step dominates the runtime (compared to tasks like storing the two sets of nodes used and nodes not yet used). Thus, the runtime will be

$$\begin{aligned} r(n) &= \sum_{i=1}^{n} i \times (n-i) \\ &= \sum_{i=1}^{n} n \cdot i - \sum_{i=1}^{n} i \cdot i \\ &= n \sum_{i=1}^{n} i - \sum_{i=1}^{n} i \cdot i \\ &= n \frac{n \cdot (n+1)}{2} - \frac{n \cdot (n+1) \cdot (2n+1)}{6} \\ &\in \Theta(n^3). \end{aligned}$$

Listing 10.1: An $O(n^3)$ naive implementation of Prim's algorithm for computing an MST.

```
import numpy
n=8

def draw(g, unweighted_mst_edges):
  import pylab as P
  pos = networkx.circular_layout(g)
  networkx.draw_networkx(g, pos=pos, with_labels=True, node_color='white')
  #print unweighted_mst_edges
  t_star = networkx.from_edgelist(unweighted_mst_edges)
  #print t_star.edges()
  networkx.draw_networkx_edges(t_star, pos=pos, node_color='white',
      width=3)
  edge_labels=dict([ (e,A[e[0],e[1]]) for e in g.edges() ])
  networkx.draw_networkx_edge_labels(g, pos=pos, edge_labels=edge_labels)
  P.show()

numpy.random.seed(0)

# create an adjacency matrix:
A = numpy.random.randint(0,100,(n,n))
for i in xrange(n):
```

```
  # make distance from i to i 0:
  A[i,i] = 0
  for j in xrange(i+1,n):
    # make distance from i to j match the distance from j to i
    # (i.e., make into an undirected graph):
    A[i,j] = A[j,i]

mst_edges = []

# start with node 0 in the MST
nodes_used = set([0])
nodes_not_used = set([ i for i in xrange(n) ])

# in each iteration, grow the include nodes to the next node connected:
while len(nodes_used) != n:
  edges_between_included_and_not_included = []

  # runtime: |nodes_used| * (n-|nodes_used|)
  for i in nodes_used:
    for j in nodes_not_used:
      edges_between_included_and_not_included.append( (A[i,j], i, j) )

  # add minimum weight edge between included and not included nodes:
  best_edge = min(edges_between_included_and_not_included)
  mst_edges.append(best_edge)

  # best edge is of form (weight, i, j) where j is a node not used:
  node_not_yet_used = best_edge[2]

  # runtime from the following lines will each be \in \Theta(n
  # \log(n)) using balanced binary tree sets:
  nodes_used.add(node_not_yet_used)
  nodes_not_used.remove(node_not_yet_used)

print 'MST cost', sum( [ e[0] for e in mst_edges ] )
print mst_edges

import networkx
g = networkx.from_numpy_matrix(A)
draw(g, [ (e[1],e[2]) for e in mst_edges ] )
```

This can be improved using a min-heap to store the current cut (thereby updating it in a sparse manner): the heap begins by holding the edges from the first node and the tree to all other nodes. The next edge is found by

popping the minimum weight edge from the heap. Then all cycle-forming edges to the node just added to the tree must be removed[2]. Likewise, we add all edges starting from the newly added node. Rebuilding a heap of at most $n \times n$ edges will cost $\in O(n^2)$ using a binary heap. Likewise, each edge insertion will cost $\in O(\log(n))$ and there will be at most n edges incident to the newly added node, so the runtime from adding new edges will cost $\in O(n \log(n))$. Thus the cost of each iteration will be dominated by the heap rebuilding, and the total runtime will be unchanged at $O(n^3)$ (Listing 10.2).

However, if we were able to surgically remove the $\leq n$ edges to the newly added node, then this would cost $O(n \log(n))$ instead of $O(n^2)$ each iteration, and so the runtime would be n iterations each with a runtime $\in O(n \log(n))$ for a total runtime of $n^2 \log(n)$.

Listing 10.2: An $O(n^3)$ heap-based implementation of Prim's algorithm for computing an MST. This could be replaced by an $O(n^2 \log(n))$ implementation of Prim's algorithm if `Python`'s heap data structure were not so (INSERT MICROAGGRESSION OF YOUR CHOICE HERE).

```
import numpy
n=8

def draw(g, unweighted_mst_edges):
  import pylab as P
  pos = networkx.circular_layout(g)
  networkx.draw_networkx(g, pos=pos, with_labels=True, node_color='white')
  t_star = networkx.from_edgelist(unweighted_mst_edges)
  networkx.draw_networkx_edges(t_star, pos=pos, node_color='white',
      width=3)
  edge_labels=dict([ (e,A[e[0],e[1]]) for e in g.edges() ])
  networkx.draw_networkx_edge_labels(g, pos=pos, edge_labels=edge_labels)
  P.show()

numpy.random.seed(0)

# create an adjacency matrix:
A = numpy.random.randint(0,100,(n,n))
for i in xrange(n):
  # make distance from i to i 0:
  A[i,i] = 0
  for j in xrange(i+1,n):
```

[2]Unfortunately, `Python`'s `heapq` module does not offer a means to remove arbitrary elements, and so the best we can do is to rebuild the heap from scratch.

```
    # make distance from i to j match the distance from j to i
    # (i.e., make into an undirected graph):
    A[i,j] = A[j,i]

import heapq

mst_edges = []

# start with node 0 in the MST
nodes_used = set([0])
nodes_not_used = set([ i for i in xrange(n) ])

cut_edges = []
for j in xrange(1,n):
  heapq.heappush(cut_edges, (A[0,j], 0, j))

# in each iteration, grow the include nodes to the next node connected:
while len(nodes_used) != n:
  best_edge = heapq.heappop(cut_edges)

  mst_edges.append(best_edge)

  # best edge is of form (weight, i, j) where j is a node not used:
  node_not_yet_used = best_edge[2]

  # runtime from the following lines will each be \in \Theta(n
  # \log(n)) using balanced binary tree sets:
  nodes_used.add(node_not_yet_used)
  nodes_not_used.remove(node_not_yet_used)

  # remove all edges to node_not_yet_used (there will be n of them, so
  # the cost will be n heap removals or n \log(n) using a binary heap):

  # python's heap desperately needs a remove function :(

  # therefore, we will rebuild the heap from scratch in O(k \log(k)),
  # where k is at most n^2:
  new_cut_edges = []
  for edge in cut_edges:
    if edge[2] != node_not_yet_used:
      new_cut_edges.append(edge)
  cut_edges = new_cut_edges
  heapq.heapify(cut_edges)

  # add new edges starting from node_not_yet_used
```

```
  for i in nodes_not_used:
    heapq.heappush(cut_edges, (A[node_not_yet_used,i], node_not_yet_used,
        i))

print 'MST cost', sum( [ e[0] for e in mst_edges ] )
print mst_edges

import networkx
g = networkx.from_numpy_matrix(A)
draw(g, [ (e[1],e[2]) for e in mst_edges ] )
```

10.1.2 Achieving an $O(n^2)$ Prim's algorithm

In the worst-case scenario, $|E| \in \Theta(n^2)$, and our previous implementations of Prim's algorithm have runtime $\in \Omega(n^2)$; however, we can improve this by exchanging our edge-centric view for a more node-centric view of the problem.

First, we will cease storing the edges in the cut between the nodes in T^* and the nodes not yet in T^*. Instead, let us simply store the shortest edge to every node not yet in T^*. When we begin, our tree T^* contains only an arbitrary node, w.l.o.g.[3], node 0. So we first fill our minimum cut edges with the edges between nodes $\{0\}$ and $\{1, 2, \ldots n-1\}$. In this manner, given any $i \neq 0$, we can find the shortest edge from our current nodes in T^* and the nodes not yet added to T^*. We will maintain this property as an invariant.

In every iteration, we will find the yet unused node $i \notin T^*$ with the shortest edge into it. This will require a pass through our n minimum cut edges. When the lowest cost edge in the cut is selected, we will add the new node to T^* using that edge. Now that i has been added to T^*, we will remove edges into i from the minimum cut edges. Likewise, we may now add any edge from i into every node $j \notin T^*$; we will add any such edges if they improve the distance over the current edge into j in the current minimum cut edges. This step only requires another $O(n)$ pass.

In this manner, each iteration is only $O(n)$, and we maintain the invariant that we always have the best edges into every node $\notin T^*$. Thus we can avoid using the entire cut set. The runtime of Prim's algorithm, which performs $n-1$ of these iterations will therefore be $\in O(n^2)$. This can be seen in Listing 10.3.

[3] "Without loss of generality" means that the proof would work with another number, *e.g.*, if we began our tree containing only node 1.

Prim's algorithm *cannot* be run substantially faster than roughly $\frac{n^2}{2}$, because it must load an input matrix of $n \times n$ distances[4] (or roughly half of those distances when the graph is undirected). For this reason, Prim's algorithm is $\in \Omega(n^2)$, and we see that it is therefore $\in \Theta(n^2)$.

Listing 10.3: An $O(n^2)$ implementation of Prim's algorithm for computing an MST. This version takes a node-centric view to improve performance when $|E| \in \Theta(n^2)$.

```
import numpy
n=8

def draw(g, unweighted_mst_edges):
  import pylab as P
  pos = networkx.circular_layout(g)
  networkx.draw_networkx(g, pos=pos, with_labels=True, node_color='white')
  #print unweighted_mst_edges
  t_star = networkx.from_edgelist(unweighted_mst_edges)
  #print t_star.edges()
  networkx.draw_networkx_edges(t_star, pos=pos, node_color='white',
      width=3)
  edge_labels=dict([ (e,A[e[0],e[1]]) for e in g.edges() ])
  networkx.draw_networkx_edge_labels(g, pos=pos, edge_labels=edge_labels)
  P.show()

numpy.random.seed(0)

# create an adjacency matrix:
A = numpy.random.randint(0,100,(n,n))
for i in xrange(n):
  # make distance from i to i 0:
  A[i,i] = 0
  for j in xrange(i+1,n):
    # make distance from i to j match the distance from j to i
    # (i.e., make into an undirected graph):
    A[i,j] = A[j,i]

mst_edges = []

# start with node 0 in the MST
nodes_used = set([0])
nodes_not_used = set([ i for i in xrange(n) ])
```

[4] Assuming the graph is dense in the case $|E| \in \Theta(n^2)$.

```
best_edge_between_used_and_each_node_not_used = [None]*n
for i in nodes_not_used:
  best_edge_between_used_and_each_node_not_used[i] = (A[0,i], (0, i))
best_edge_between_used_and_each_node_not_used[0] = (numpy.inf, (-1, -1))

# in each iteration, grow the include nodes to the next node connected:
# each iteration costs O(n), so the total cost is in O(n^2).
while len(nodes_used) != n:
  best_dist, (node_used, node_not_yet_used) =
      min(best_edge_between_used_and_each_node_not_used)
  mst_edges.append( (A[node_used, node_not_yet_used], node_used,
      node_not_yet_used) )

  # runtime from the following lines will each be \in \Theta(n
  # \log(n)) using balanced binary tree sets:
  nodes_used.add(node_not_yet_used)
  nodes_not_used.remove(node_not_yet_used)

  # add edges from node_not_yet_used to all the remaining unused nodes:
  for i in nodes_not_used:
    if A[node_not_yet_used, i] <
        best_edge_between_used_and_each_node_not_used[i][0]:
      best_edge_between_used_and_each_node_not_used[i] =
          (A[node_not_yet_used,i], (node_not_yet_used, i))

  best_edge_between_used_and_each_node_not_used[node_not_yet_used] =
      (numpy.inf, (-1, -1))

print 'MST cost', sum( [ e[0] for e in mst_edges ] )
print mst_edges

import networkx
g = networkx.from_numpy_matrix(A)
draw(g, [ (e[1],e[2]) for e in mst_edges ] )
```

10.1.3 Kruskall's algorithm

The cost of Kruskall's algorithm will be the cost of sorting the edges by weight and then the cost of linearly progressing through the edges. Assuming our sort is comparison based, we know that our optimal runtime of sort will be $\in \Theta(|E| \cdot \log(|E|))$, where $|E|$ is the number of edges (proven in Chapter 6) plus our progression through every edge in ascending order of weight, where

we must investigate whether each new edge would introduce a cycle. This is resolved using a "disjoint set" data structure, also known as a "union-find" data structure.

This disjoint data structure allows one to merge sets and to find the "leader" object of every disjoint set (which is called the "root" of the set). Disjoint sets are slightly tricky to understand, but they fit the use-case of searching for edges that would introduce cycles so well that they are essentially defined for this precise problem. A balanced, path-compressed implementation of a disjoint set data structure is implemented in Listing 10.4. The runtime of each `union` and each `find` operation will be very efficient: both will be in $\widetilde{O}(\log^*(n))$, where $\log^*$ denotes the iterative logarithm, *i.e.*, the number of logarithms that must be applied before n becomes 1. $log^*(n)$ is so efficient that even for enormous values of n, $log^*(n)$ is bounded above by a small constant of around 5[5]. For example, the number of particles in the universe, $n \approx 2^{270}$ would have a $log^*(n) < 5$. Note that we have not proven that amortized runtime here, so just take it on faith for the moment.

Listing 10.4: A disjoint set data structure where each operation is $\in \widetilde{O}(\log^*(n))$.

```
class DisjointSet():
  # create one with a unique value:
  def __init__(self, val):
    self.value = val
    self.parent = self

    # attribute is only valid at root nodes:
    self.rank = 1

  def find_root(self):
    if self.parent != self:
      # path compression:
      # when finding the root, flatten the tree as
      # you go up so that the distance to the root will shrink:
      root = self.parent.find_root()
      self.parent = root
      return root
    return self

  def union_with(self, rhs):
```

[5]However, this is only a practical statement of performance and we still *cannot* treat $\log^*(n)$ as a constant when we consider asymptotic runtimes in big-oh and big-Ω bounds.

```
    root = self.find_root()
    rhs_root = rhs.find_root()

    if root != rhs_root:
      # not in the same tree, need to unite:

      if root.rank < rhs_root.rank:
        # add self to rhs's tree:
        root.parent = rhs_root
      elif root.rank > rhs_root.rank:
        rhs_root.parent = root
        # add rhs to self's tree:
      else:
        # add rhs to self's tree and increase rank:
        rhs_root.parent = root
        root.rank += 1
```

Because the disjoint set data structure will be so efficient, the runtime of Kruskall's algorithm will be dominated by sorting the edges: $r(n) \in O(|E| \cdot \log(|E|) + |E| \cdot \log^*(|E|)) = O(|E| \cdot \log(|E|))$. In the worst-case scenario the graph defines all possible edges: $|E| \in \Theta(n^2)$. In that case we have runtime $\in O(n^2 \log(n^2)) = O(n^2 \log(n))$. Kruskall's algorithm is shown in Listing 10.5.

Listing 10.5: An $O(n^2 \log(n))$ implementation of Kruskall's algorithm for computing an MST.

```
import numpy
from disjoint_set import *
n=8

def draw(g, unweighted_mst_edges):
  import pylab as P
  pos = networkx.circular_layout(g)
  networkx.draw_networkx(g, pos=pos, with_labels=True, node_color='white')
  t_star = networkx.from_edgelist(unweighted_mst_edges)
  networkx.draw_networkx_edges(t_star, pos=pos, node_color='white',
      width=3)
  edge_labels=dict([ (e,A[e[0],e[1]]) for e in g.edges() ])
  networkx.draw_networkx_edge_labels(g, pos=pos, edge_labels=edge_labels)
  P.show()

numpy.random.seed(0)
```

```
# create an adjacency matrix:
A = numpy.random.randint(0,100,(n,n))
for i in xrange(n):
  # make distance from i to i 0:
  A[i,i] = 0
  for j in xrange(i+1,n):
    # make distance from i to j match the distance from j to i
    # (i.e., make into an undirected graph):
    A[i,j] = A[j,i]

edges = [ (A[i,j],i,j) for i in xrange(n) for j in xrange(n) if i != j ]
ascending_edges = sorted(edges)

mst_edges = []

connected_graphs = [ DisjointSet(i) for i in xrange(n) ]
for e in ascending_edges:
  weight,i,j = e
  if connected_graphs[i].find_root() != connected_graphs[j].find_root():
    # this edge does not introduce a loop, it can be added:
    mst_edges.append(e)
    connected_graphs[i].union_with(connected_graphs[j])

print 'MST cost', sum( [ e[0] for e in mst_edges ] )
print mst_edges

import networkx
g = networkx.from_numpy_matrix(A)
draw(g, [ (e[1],e[2]) for e in mst_edges ] )
```

10.2 The Traveling salesman problem

A "traveling salesman problem" (TSP) defines the task of finding the Hamiltonian path with minimum total weight. A Hamiltonian path is a path[6] that visits each node in the graph exactly once. We will denote this minimum weight Hamiltonian path as P^*.

Interestingly, even though TSP is highly similar to MST, it is currently unknown whether an efficient exact solution to TSP is even possible on an arbitrary graph.

[6] A path is a tree that doesn't fork. Paths will always be of the form $a \to b \to \cdots$

10.2.1 Solving with brute force

One solution to TSP is to simply try all possible $n!$ orderings of vertices in the graph and for each ordering, finding the cost of its corresponding path (Listing 10.6).

Listing 10.6: An $O(n!)$ brute-force solution to TSP. The resulting path costs 108.

```
import numpy
n=8

def draw(g, unweighted_mst_edges):
  import pylab as P
  pos = networkx.circular_layout(g)
  networkx.draw_networkx(g, pos=pos, with_labels=True, node_color='white')
  t_star = networkx.from_edgelist(unweighted_mst_edges)
  networkx.draw_networkx_edges(t_star, pos=pos, node_color='white',
      width=3)
  edge_labels=dict([ (e,A[e[0],e[1]]) for e in g.edges() ])
  networkx.draw_networkx_edge_labels(g, pos=pos, edge_labels=edge_labels)
  P.show()

numpy.random.seed(0)

# create an adjacency matrix:
A = numpy.random.randint(0,100,(n,n))
for i in xrange(n):
  # make distance from i to i 0:
  A[i,i] = 0
  for j in xrange(i+1,n):
    # make distance from i to j match the distance from j to i
    # (i.e., make into an undirected graph):
    A[i,j] = A[j,i]

# make the graph metric:
for i in xrange(n):
  for j in xrange(n):
    for k in xrange(n):
      A[i,j] = min(A[i,j], A[i,k] + A[k,j])

import itertools
all_paths = itertools.permutations(numpy.arange(n))

best_path=None
```

```
best_path_score=numpy.inf

def score(path):
  result = 0
  for i in xrange(len(path)-1):
    edge = (path[i], path[i+1])
    result += A[edge]
  return result

# there will be n! of these:
for path in all_paths:
  path_score = score(path)
  if path_score < best_path_score:
    best_path = path
    best_path_score = path_score

print 'TSP cost', best_path_score
print best_path

best_path_edges = [ (best_path[i], best_path[i+1]) for i in xrange(n-1) ]

import networkx
g = networkx.from_numpy_matrix(A)
draw(g, best_path_edges)
```

10.2.2 2-Approximation

In cases when n is small, brute force is straightforward and will not be too inefficient; however, when n is large[7], brute force is infeasibly slow. For this reason, we may be satisfied with a *decent* solution rather than an optimal solution that is perpetually out of reach. We can arrive at a proveably decent solution, an approximation, to TSP, by using MST.

First we observe that $paths(G) \subset trees(G)$, implying that the best tree through all nodes is as good or better than the best path through all nodes. Thus $cost(T^*) \leq cost(P^*)$, *i.e.*, the cost of the MST will be as good or better than the cost of the TSP solution; however, this may be uninteresting for many cases, because the MST is not a path (as required by the solution to TSP). For this reason, we will turn our MST solution into a path.

First consider a depth-first search (DFS) through every node in T^*. We

[7] $60! > 2^{270}$, meaning that 60! will be larger than the number of particles in the universe.

can turn trees with lots of forks into a path as follows: At every node a with adjacent nodes $b_1, b_2, b_3, \ldots$, we can visit the nodes in the order $a \to b_1 \to a \to b_2 \to a \to b_3 \to \cdots$. This path P will pass through every edge exactly twice and thus the cost of this path will be no more than $2cost(T^*)$. This is still not a valid Hamiltonian path, because it may return to nodes multiple times. In some cases, this may be acceptable, but not always (*e.g.*, what if the "cities" modeled by each vertex are destroyed after we visit them?).

Here we can employ a property of "metric" graphs: If our graph is "metric" than the distance $\forall c, E_{a,b} \leq E_{a,c} + E_{c,b}$, meaning the distance between nodes a and b cannot be improved by passing through another node. If our graph is metric, then every collection of edges $b_1 \to a \to b_2$ can be replaced with $b_1 \to b_2$, thereby avoiding visiting a multiple times and by taking a more direct route (which cannot make the path more costly in the case of a metric graph). By doing this, we can convert our MST to a Hamiltonian path $P^{(m)}$.

$$\begin{aligned} cost(P^{(m)}) &\leq cost(P) \\ &\leq 2cost(T^*) \\ &\leq 2cost(P^*) \end{aligned}$$

Thus on a metric graph, we have used MST to achieve a 2-approximation of TSP. This means that we will efficiently[8] generate a Hamiltonian path with cost $\leq cost(P^*)$. Either of our MST implementations can be used to achieve this result. Listing 10.7 demonstrates this 2-approximation.

Listing 10.7: A 2-approximation to TSP via MST. The TSP cost was 108 (Listing 10.6). The cost of the MST is 87 (less than the or equal to the cost of the TSP solution, as proven), and the cost of the 2-approximation path is 130 (less than or equal to twice the TSP cost, as proven).

```
import numpy
n=8

def draw(g, unweighted_mst_edges):
  import pylab as P
  pos = networkx.circular_layout(g)
  networkx.draw_networkx(g, pos=pos, with_labels=True, node_color='white')
  t_star = networkx.from_edgelist(unweighted_mst_edges)
```

[8] Much more efficiently than the $O(n!)$ required by brute force.

```
  networkx.draw_networkx_edges(t_star, pos=pos, node_color='white',
      width=3)
  edge_labels=dict([ (e,A[e[0],e[1]]) for e in g.edges() ])
  networkx.draw_networkx_edge_labels(g, pos=pos, edge_labels=edge_labels)
  P.show()

numpy.random.seed(0)

# create an adjacency matrix:
A = numpy.random.randint(0,100,(n,n))
for i in xrange(n):
  # make distance from i to i 0:
  A[i,i] = 0
  for j in xrange(i+1,n):
    # make distance from i to j match the distance from j to i
    # (i.e., make into an undirected graph):
    A[i,j] = A[j,i]

# make the graph metric:
for i in xrange(n):
  for j in xrange(n):
    for k in xrange(n):
      A[i,j] = min(A[i,j], A[i,k] + A[k,j])

mst_edges = []

# start with node 0 in the MST
nodes_used = set([0])
nodes_not_used = set([ i for i in xrange(n) ])

best_edge_between_used_and_each_node_not_used = [None]*n
for i in nodes_not_used:
  best_edge_between_used_and_each_node_not_used[i] = (A[0,i], (0, i))
best_edge_between_used_and_each_node_not_used[0] = (numpy.inf, (-1, -1))

# in each iteration, grow the include nodes to the next node connected:
# each iteration costs O(n), so the total cost is in O(n^2).
while len(nodes_used) != n:
  best_dist, (node_used, node_not_yet_used) =
      min(best_edge_between_used_and_each_node_not_used)
  mst_edges.append( (A[node_used, node_not_yet_used], node_used,
      node_not_yet_used) )

  # runtime from the following lines will each be \in \Theta(n
  # \log(n)) using balanced binary tree sets:
```

```
  nodes_used.add(node_not_yet_used)
  nodes_not_used.remove(node_not_yet_used)

  # add edges from node_not_yet_used to all the remaining unused nodes:
  for i in nodes_not_used:
    if A[node_not_yet_used, i] <
        best_edge_between_used_and_each_node_not_used[i][0]:
      best_edge_between_used_and_each_node_not_used[i] =
          (A[node_not_yet_used,i], (node_not_yet_used, i))

  best_edge_between_used_and_each_node_not_used[node_not_yet_used] =
      (numpy.inf, (-1, -1))

print 'MST cost', sum( [ e[0] for e in mst_edges ] )

def dfs_traversal(root, nodes_visited, adjacency_dict):
  if root in nodes_visited:
    return []
  nodes_visited.add(root)

  result = []
  for adjacent in adjacency_dict[root]:
    if adjacent not in nodes_visited:
      result.append( (root, adjacent) )
      result.extend(dfs_traversal(adjacent, nodes_visited,
          adjacency_dict))
      result.append( (adjacent, root) )
  return result

def compress_path(loopy_path):
  nodes_visited = set()
  nodes_in_order = []
  for edge in loopy_path:
    a, b = edge
    if a not in nodes_visited:
      nodes_in_order.append(a)
    if b not in nodes_visited:
      nodes_in_order.append(b)
    nodes_visited.add(a)
    nodes_visited.add(b)

  result = []
  for i in xrange(len(nodes_in_order)-1):
    result.append( (nodes_in_order[i], nodes_in_order[i+1]) )
```

```
  return result

def mst_edges_to_tsp_path(weighted_edges):
  adjacency_dict = {}
  for e in weighted_edges:
    weight, a, b = e
    if a not in adjacency_dict:
      adjacency_dict[a] = []
    if b not in adjacency_dict:
      adjacency_dict[b] = []

    adjacency_dict[a].append(b)
    adjacency_dict[b].append(a)

  loopy_path = dfs_traversal(0, set(), adjacency_dict)
  tsp_path = compress_path(loopy_path)
  return tsp_path

def score(path_edges):
  result = 0
  for edge in path_edges:
    result += A[edge]
  return result

tsp_2_approx_path = mst_edges_to_tsp_path(mst_edges)
print tsp_2_approx_path
print '2 approx cost', score(tsp_2_approx_path)

import networkx
g = networkx.from_numpy_matrix(A)
#draw(g, [ (e[1],e[2]) for e in mst_edges ] )
draw(g, tsp_2_approx_path )
```

Note that this 2-approximation may, in some cases, be improved by using this as a "seed" for branch and bound: if the initial MST is good, it may help eliminate many of the paths investigated by subsequent brute force, even without fully creating a brute force configuration. For example, if the MST-based 2-approximation achieves a Hamiltonian path with total cost 9, then any solution containing the partial solution $q = a \to b \to c$ with $cost(q) > 9$ *cannot* be the optimal solution (because it already has cost higher than the Hamiltonian path from our 2-approximation and additional edges, which must have nonnegative weight, can only make the cost higher). In this manner, branch and bound can avoid visiting every configuration,

and the better the seed, the higher performance branch and bound will be. Thus our 2-approximation is also useful in speeding up brute force to achieve a faster optimal solution.

When our initial adjacency matrix is *not* metric[9], we can convert the adjacency matrix into a metric matrix by solving the "all-pairs shortest paths" (APSP) problem. APSP finds, for each pair of nodes a and b, the lowest cost path $p_{a,b}$ that starts at a and ends at b. If we replace our initial adjacency matrix E with E', the adjacency matrix after performing APSP, then we are guaranteed that $E'_{a,b} \leq E'_{a,c} + E'_{c,b}$, because every path through c must have been considered when finding the minimum APSP distances, E'.

Using APSP to force a non-metric graph to behave like a metric graph will not be applicable in all cases. As before, if we absolutely need a true Hamiltonian path, then the MST-based 2-approximation of TSP can only be used if the original graph is metric.

[9]We can easily verify this in $O(n^3)$ time by checking that the distance between every pair of nodes a and b *cannot* be short cut by passing through a third node c: $\forall a \forall b \forall c, E_{a,c} + E_{c,b} \geq E_{a,b}$.

Chapter 11

Gauss and Karatsuba Multiplication

11.1 Multiplication of complex numbers

As we remember from grade school, addition is fairly easy, while multiplication is more difficult. This coincides with an algorithmic analysis of addition and grade-school multiplication: Adding two n-digit numbers performs n operations on digit pairs[1], up to n carry operations, and is thus $\in O(n)$. Grade-school multiplication, on the other hand, pairs each digit from one number with every digit from the other number. Thus, even before these products are totaled, we have $n \times n$ operations, and we can see that the grade-school multiplication algorithm is $\in \Omega(n^2)$.

Historically, this meant that scientists doing computations by hand would gladly perform a few additions instead of a multiplication between two large numbers. This was the case with the scientist Gauß [2]: while working with complex numbers, Gauß frequently performed complex multiplications

$$(a + b \cdot j) \cdot (c + d \cdot j) = a \cdot c - b \cdot d + (a \cdot d + b \cdot c) \cdot j.$$

where $j = \sqrt{-1}$. This complex product can, of course, be found using four real products, $a \cdot c$, $b \cdot d$, $a \cdot d$, and $b \cdot c$. This is implemented in Listing 11.1.

[1] Aligned so that the 10's place of one number is adding to the 10's place of the other number, *etc.*

[2] My great-great-great-$\cdots$ grand advisor[3]

[3] Or Groß-groß-$\cdots$ Doktorvater

Listing 11.1: Multiplying complex numbers using the naive approach, with four multiplications.

```
import numpy

class Complex:
  def __init__(self, real, imag=0.0):
    self.real = real
    self.imag = imag

  def __add__(self, rhs):
    return Complex(self.real+rhs.real, self.imag+rhs.imag)

  def __sub__(self, rhs):
    return Complex(self.real-rhs.real, self.imag-rhs.imag)

  def __mul__(self, rhs):
    return Complex(self.real*rhs.real-self.imag*rhs.imag,
        self.real*rhs.imag+self.imag*rhs.real)

  def __str__(self):
    result=''
    if numpy.fabs(self.real) > 0.0 and numpy.fabs(self.imag) > 0.0:
      return str(self.real) + '+' + str(self.imag) + 'j'
    if numpy.fabs(self.real) > 0.0:
        return str(self.real)
    if numpy.fabs(self.imag) > 0.0:
      return str(self.imag) + 'j'
    return '0'

x=Complex(1,2)
y=Complex(2,3)

print x, y, x*y
```

At first glance, it may appear that there is no way to reduce the number of multiplications: every term in the first complex number *must* be multiplied with every term in the second complex number; however, because we are working on a "ring" space[4], we can also add values, multiply them, and then subtract out. In this manner, we could potentially multiply some merged values and then subtract others out in order to get the result we want.

For example, if we wanted to compute $(a + b) \cdot (c + d)$, we could do so

[4]We are working on a "ring" whenever we are working on a set of objects that support addition and also support its inverse operation, subtraction.

	a	$b \cdot j$
c	$a \cdot c$	$b \cdot c \cdot j$
$d \cdot j$	$a \cdot d \cdot j$	$-b \cdot d$

Table 11.1: The four terms in a complex multiplication.

with the naive approach (using four multiplications), but we could also do so by computing $e = a + b$ and $f = c + d$, and then computing $e \cdot f$ (using one multiplication). In a more sophisticated use of this approach, we could compute $(a+b)\cdot c$, $(a+b)\cdot d$, and $(a+b)\cdot(c+d)$ by first computing $e = (a+b)\cdot c$ and $f = (a + b) \cdot d$, and then computing $(a + b) \cdot (c + d) = e + f$. These approaches only add, but do not subtract, and thus they do not yet need the ring property.

Let us consider the four products that we perform when doing our complex multiplication (Table 11.1). The results of the complex multiplication will be the sum of terms on the diagonal (these will form the real result) and the sum of terms positive diagonal (these will form the complex result). Consider what would happen if we compute $e = (a+b\cdot j)\cdot c$ and $f = (c+d\cdot j)\cdot a$. The value of e will compute the sum of the first row in Table 11.1, while f will compute the sum of the first column in Table 11.1; therefore, we can use the ring property to subtract and compute the difference of terms on the positive diagonal. This will remove the top left cell of Table 11.1, leaving behind only contributions from the top right and bottom left cells in the table:

$$\begin{aligned} e - f &= a \cdot c + b \cdot c \cdot j - c \cdot a - d \cdot a \cdot j \\ &= b \cdot c \cdot j - d \cdot a \cdot j. \end{aligned}$$

However, consider that the imaginary result of the complex multiplication will be $b \cdot c \cdot j + d \cdot a \cdot j$; for this reason, we instead use $e = (a + b \cdot j) \cdot c$ and $f = (c - b \cdot j) \cdot a$. Thus we invert the sign of that second term:

$$\begin{aligned} e - f &= a \cdot c + b \cdot c \cdot j - c \cdot a + d \cdot a \cdot j \\ &= b \cdot c \cdot j + d \cdot a \cdot j. \end{aligned}$$

An important next step is removing whether values are real or imaginary so that the multiplication can be done on merged values. Specifically, if we let

$e = (a + b) \cdot c$ and $f = (c - d) \cdot a$, then we have

$$\begin{aligned} e - f &= a \cdot c + b \cdot c - c \cdot a + d \cdot a \\ &= b \cdot c + d \cdot a, \end{aligned}$$

which is the coefficient of the imaginary term in the result (*i.e.*, we need only multiply $e - f$ with j and we get the imaginary term). In this manner, we reduce the total number of multiplications to two instead of four when computing e and f.

Of course, this does not yet confer an advantage over the naive approach: we have used two multiplications (one each to get e and f), and we could compute the imaginary result using the naive approach with two multiplications (and no subtraction). But consider that in computing e and f in this way, that we have already computed the sum of the first row, e. If we compute the sum of terms in the second column, we could perhaps do the same as what we did above, and use the sum of the first row and the sum of the second column to subtract out and remove the top right cell of Table 11.1. Given $e = (a + b) \cdot c$, we let $g = (c + d) \cdot b$ and proceed as we did above:

$$\begin{aligned} e - g &= a \cdot c + b \cdot c - c \cdot b - d \cdot b \\ &= a \cdot c - b \cdot d. \end{aligned}$$

We can compute e, f, and g in three multiplications, and thus we compute the real part of the complex multiplication as $e - f$ and the imaginary part of the complex multiplication as $e - g$. In this manner, we use only three multiplications, although we need to introduce some additions and subtractions. Gauß actually discovered and used this approach when performing complex multiplications on large numbers; the extra cost of the linear-time additions and subtractions (of which there are a constant number) was well worth avoiding a large grade-school multiplication between to large numbers. This is implemented in Listing 11.2.

Listing 11.2: Multiplying complex numbers using Gauß multiplication, with three multiplications instead of four.

```
import numpy

class Complex:
  def __init__(self, real, imag=0.0):
    self.real = real
```

```
    self.imag = imag

  def __add__(self, rhs):
    return Complex(self.real+rhs.real, self.imag+rhs.imag)

  def __sub__(self, rhs):
    return Complex(self.real-rhs.real, self.imag-rhs.imag)

  def __mul__(self, rhs):
    # (a+bi)*(c+di)

    # a*c  a*d = a*(c+d) = x
    # b*c -b*d = b*(c-d) = z
    # =
    # c*(b-a)
    # =
    # y

    # x=a*(c+d)
    # y=c*(b-a)
    # z=b*(c-d)
    # imag=x+y
    # real=z-y

    x = self.real*(rhs.real+rhs.imag)
    y = rhs.real*(self.imag-self.real)
    z = self.imag*(rhs.real-rhs.imag)
    return Complex(z-y, x+y)

  def __str__(self):
    result=''
    if numpy.fabs(self.real) > 0.0 and numpy.fabs(self.imag) > 0.0:
      return str(self.real) + '+' + str(self.imag) + 'j'
    if numpy.fabs(self.real) > 0.0:
       return str(self.real)
    if numpy.fabs(self.imag) > 0.0:
      return str(self.imag) + 'j'
    return '0'

x=Complex(1,2)
y=Complex(2,3)

print x, y, x*y
```

	$x_{\text{high}} \cdot 10^{\frac{n}{2}}$	x_{low}
$y_{\text{high}} \cdot 10^{\frac{n}{2}}$	$x_{\text{high}} \cdot y_{\text{high}} \cdot 10^{n}$	$x_{\text{low}} \cdot y_{\text{high}} \cdot 10^{\frac{n}{2}}$
y_{low}	$x_{\text{high}} \cdot y_{\text{low}} \cdot 10^{\frac{n}{2}}$	$x_{\text{low}} \cdot y_{\text{low}}$

Table 11.2: The four terms in a recursive integer multiplication.

11.2 Fast multiplication of long integers

As impressive as it is, Gauß trick does not make the multiplications between large integers any easier. The principal trouble of this is that Gauß complex multiplication trick does not recurse: it reduces a complex multiplication to three real multiplications (and some additions and subtractions).

In the mid 20^{th} century, the famous scientist Kolmogorov (now famous for Kolmogorov complexity and the Kolmogorov-Smirnov test) conjectured that multiplication between two n-digit integers was $\in \Omega(n^2)$, that essentially there was no method substantially better than the grade-school algorithm, wherein every digit of one number touches every digit of the other number. At place where he presented this conjecture, a young man named Karatsuba was in the audience. Karatsuba tried to attack this problem and disproved Kolmogorov's conjecture, and in doing so, Karatsuba discovered the first algorithm for multiplying two n-digit numbers $\in o(n^2)$.

The trick behind Karatsuba's algorithm was essentially to use Gauß algorithm, but instead of multiplying two complex numbers, Karatsuba used it to multiply two n-digit integers. In this manner, he reduced the product of two n-digit integers to smaller problems of the same type (whereas Gauß had reduced complex multiplication to real multiplication, a different problem).

Consider the product between two integers, x and y. We can split x into its most-significant digits x_{high} and its less-significant digits x_{low}. In this manner $x = x_{\text{high}} 10^{\frac{n}{2}} + x_{\text{low}}$, where x has n decimal digits[5]:

$$\begin{aligned} x \cdot y &= (x_{\text{high}} \cdot 10^{\frac{n}{2}} + x_{\text{low}}) \cdot (y_{\text{high}} \cdot 10^{\frac{n}{2}} + y_{\text{low}}) \\ &= x_{\text{high}} \cdot y_{\text{high}} \cdot 10^{n} + (x_{\text{high}} \cdot y_{\text{low}} + x_{\text{low}} \cdot y_{\text{high}}) \cdot 10^{\frac{n}{2}} + x_{\text{low}} \cdot y_{\text{low}}. \end{aligned}$$

Table 11.2 rewrites the table from the Gauß multiplication.

This will produce the runtime recurrence $r(n) = 4r\left(\frac{n}{2}\right) + \Theta(n)$. Using the master theorem, we see that this is leaf-heavy with $r(n) \in \Theta(n^{\log_2(4)}) =$

[5]For convenience, assume that n is divisible by 2.

$\Theta(n^2)$; it realizes but does not improve over the grade-school multiplication method.

To compute the integer product, we need to compute $x_{\text{high}} \cdot y_{\text{high}}$ (which will occur at the signifcance 10^n) and $x_{\text{low}} \cdot y_{\text{low}}$ (which will occur at the significance 10^0). The remaining two terms both have significance $10^{\frac{n}{2}}$. Unlike Gauß trick, there is no need for a sign change, which simplifies things. We simply compute three products: First, $z_{\text{high}} = x_{\text{high}} \cdot y_{\text{high}}$ and $z_{\text{low}} = x_{\text{low}} \cdot y_{\text{low}}$ compute the highest and lowest significances in the result. Lastly, we compute the sum of all terms in Table 11.2 (with their significances stripped away), $z_{\text{total}} = (x_{\text{high}} + x_{\text{low}}) \cdot (y_{\text{high}} + y_{\text{low}})$. The medium-significance terms of the result (which occur on the positive diagonal of Table 11.2) can be recovered by subtracting $z_{\text{medium}} = z_{\text{total}} - z_{\text{high}} - z_{\text{low}}$.

Thus, the low-, medium-, and high-significance parts of $z = x \cdot y$ can be computed with three multiplications. We reassemble these into $z = z_{\text{high}} \cdot 10^n + z_{\text{medium}} \cdot 10^{\frac{n}{2}} + z_{\text{low}}$. Karatsuba's method reduces an n-digit multiplication to three $\frac{n}{2}$-digit multiplications and a constant number of additions and subtractions; therefore, we have $r(n) = 3r(\frac{n}{2}) + \Theta(n)$, which is $\in \Theta(n^{\log_2(3)}) = \Theta(n^{1.585\ldots})$ using the leaf-heavy case of the master theorem. An implementation of an arbitrary-precision integer class (with implementations of both naive $\Omega(n^2)$ and Karatsuba $\Theta(n^{1.585\ldots})$ multiplication) is implemented in Listing 11.3. The implementation has a poor runtime constant, but on large problems, the Karatsuba method will become substantially faster than the naive approach.

Listing 11.3: An arbitrary-precision binary integer class. Naive multiplication is implemented in an $\Omega(n^2)$ manner, while fast Karatsuba multiplication is implemented in $\Theta(n^{1.585\ldots})$.

```
import numpy

def carry(array):
  n = len(array)
  for i in xrange(n-1):
    # we add numbers right to left:
    j=n-i-1

    if array[j] > 1:
      array[j-1] += array[j] / 2
      array[j] = array[j] % 2

class BigInt:
```

```
def __init__(self, bitstring):
  self.bitstring = list(bitstring)

  for x in self.bitstring:
    assert(x in (0,1))

  if len(self.bitstring) == 0:
    self.bitstring = [0]

def trim_unneeded_bits(self):
  # remove leading [0,0,0,...] bits
  # e.g., [0,0,1,0,1] --> [1,0,1]
  n = len(self.bitstring)
  for i in xrange(n):
    if self.bitstring[i] == 1:
      break

  # do not trim if the value is [0].
  if self.bitstring != [0]:
    self.bitstring = self.bitstring[i:]

def __add__(self, rhs):
  n=max(len(self.bitstring), len(rhs.bitstring))
  result=[0]*(1+n)

  for i in xrange(n):
    # we add numbers right to left:
    if i < len(self.bitstring):
      result[len(result)-i-1] += self.bitstring[len(self.bitstring)-i-1]
    if i < len(rhs.bitstring):
      result[len(result)-i-1] += rhs.bitstring[len(rhs.bitstring)-i-1]

  carry(result)
  res_big_int = BigInt(result)
  res_big_int.trim_unneeded_bits()
  return res_big_int

def __sub__(self, rhs):
  rhs_comp = rhs.twos_complement(len(self.bitstring))
  result = self + rhs_comp

  result.bitstring[0] = 0
  result.trim_unneeded_bits()

  return result
```

```
  def shifted_left(self, num_bits):
    return BigInt(self.bitstring + ([0]*num_bits))

  def twos_complement(self, num_bits):
    padded = ([0]*(num_bits - len(self.bitstring))) + self.bitstring
    inverted = BigInt(padded)

    # flip bits:
    for i in xrange(len(inverted.bitstring)):
      if inverted.bitstring[i] == 0:
        inverted.bitstring[i] = 1
      else:
        inverted.bitstring[i] = 0

    # add 1:
    result = inverted + BigInt([1])
    return result

  def __str__(self):
    bin_string = ''.join([str(b) for b in self.bitstring])
    return bin_string + ' = '+ str(int(bin_string,2))

  # note: this cheats by letting each bit be >2 (they essentially use
  # python's native arbitrary precision integers). it is still \in
  # \Theta(n^2), but will have a decent runtime constant. it exists
  # for testing only.
def naive_mult(lhs, rhs):
  result = [0]*(len(lhs.bitstring)+len(rhs.bitstring))
  for i in xrange(len(lhs.bitstring)):
    for j in xrange(len(rhs.bitstring)):
      result[1+i+j] += lhs.bitstring[i]*rhs.bitstring[j]

  carry(result)
  result = BigInt(result)
  result.trim_unneeded_bits()
  return result

# r(n) = 4 r(n/2) + \Theta(n)
# --> r(n) \in \Theta(n^log_2(4)) = \Theta(n^2)
def recursive_mult(lhs, rhs):
  n = max(len(lhs.bitstring), len(rhs.bitstring))
  if len(lhs.bitstring) < len(rhs.bitstring):
    lhs = BigInt(([0]*(n - len(lhs.bitstring))) + lhs.bitstring)
  if len(rhs.bitstring) < len(lhs.bitstring):
```

```
    rhs = BigInt(([0]*(n - len(rhs.bitstring))) + rhs.bitstring)

  if n <= 2:
    return naive_mult(lhs, rhs)

  x_high = BigInt(lhs.bitstring[:n/2])
  x_low = BigInt(lhs.bitstring[n/2:])

  y_high = BigInt(rhs.bitstring[:n/2])
  y_low = BigInt(rhs.bitstring[n/2:])

  result = BigInt([0])

  z_low = recursive_mult(x_low, y_low)
  z_high = recursive_mult(x_high, y_high)
  t1 = recursive_mult(x_low, y_high)
  t2 = recursive_mult(x_high, y_low)
  z_mid = t1 + t2

  result += z_low
  msbs = n/2
  lsbs = n-n/2
  result += z_mid.shifted_left(lsbs)
  result += z_high.shifted_left(2*lsbs)

  return result

# r(n) = 3 r(n/2) + \Theta(n)
# --> r(n) \in \Theta(n^log_2(3)) = \Theta(n^1.585...)
def karatsuba_mult(lhs, rhs):
  n = max(len(lhs.bitstring), len(rhs.bitstring))
  if len(lhs.bitstring) < len(rhs.bitstring):
    lhs = BigInt(([0]*(n - len(lhs.bitstring))) + lhs.bitstring)
  if len(rhs.bitstring) < len(lhs.bitstring):
    rhs = BigInt(([0]*(n - len(rhs.bitstring))) + rhs.bitstring)

  if n <= 2:
    return naive_mult(lhs, rhs)

  x_high = BigInt(lhs.bitstring[:n/2])
  x_low = BigInt(lhs.bitstring[n/2:])

  y_high = BigInt(rhs.bitstring[:n/2])
  y_low = BigInt(rhs.bitstring[n/2:])
```

```
  result = BigInt([0])

  z_low = karatsuba_mult(x_low, y_low)
  z_high = karatsuba_mult(x_high, y_high)
  z_mid = karatsuba_mult(x_low + x_high, y_low + y_high) - z_low - z_high

  result += z_low
  msbs = n/2
  lsbs = n-n/2
  result += z_mid.shifted_left(lsbs)
  result += z_high.shifted_left(2*lsbs)

  return result

n=64
numpy.random.seed(0)
x=BigInt(numpy.random.randint(0,2,n))
y=BigInt(numpy.random.randint(0,2,n))
#x=BigInt([1,1,1])
#y=BigInt([1,0,0])
print 'x', x
print 'y', y
print ''

print 'recursive prod', recursive_mult(x, y)
print 'karatsuba prod', karatsuba_mult(x, y)
```

Even when implementing circuits that will multiply constant-precision integers (*e.g.*, creating the hardware on the CPU that will multiply two 32-bit `unsigned int` types from `C/C++`), algorithms like Karatsuba's method are important for creating circuits that implement practically fast multiplication between constant-precision integers without using a lot of silicon[6].

11.3 Practice and discussion questions

1. Prove the runtime of Karatsuba multiplication using the master theorem.

[6]Building a chip that simultaneously computes all n^2 digit products would use more silicon (which would cost substantially more money and be less power efficient) and may still not be as fast as a circuit using a divide-and-conquer strategy reminiscent of Karatsuba's method.

2. Consider $x = 1731$, $y = 8925$, and $z = x \cdot y$.

   (a) What are x_{high}, x_{low}, y_{high}, and y_{low}?

   (b) Compute z_{total} by summing and multiplying.

   (c) Compute z_{low} by multiplying. Use naive multiplication.

   (d) Compute z_{high} by multiplying. Use naive multiplication.

   (e) Compute z_{medium} by subtracting.

   (f) Compute z from z_{high}, z_{medium}, and z_{low}. Verify that this matches a naive computation of z.

3. Consider $x = 81185216$, $y = 34327283$, and $z = x \cdot y$. Assume that multiplications with fewer than 4 digits can be done using hardware. List all multiplications that will be performed using Karatsuba's recursive method.

4. Using the results in the previous question, find z using Karatsuba's method. Verify against a naive computation.

11.4 Answer key

1. $r(n) = 3r\left(\frac{n}{2}\right) + \Theta(n)$ has $\ell = n^{\log_2(3)} = n^{1.585\ldots}$ has a larger power than $f(n) \in \Theta(n)$; therefore, it is leaf heavy and the master theorem tells us that $r(n) \in \Theta(n^{\log_2(3)})$.

2. (a) $x_{\text{high}} = 17$, $x_{\text{low}} = 31$, $y_{\text{high}} = 89$, and $y_{\text{low}} = 25$.

   (b) $z_{\text{total}} = (17 + 31) \cdot (89 + 25) = 5472$

   (c) $z_{\text{high}} = 17 \cdot 89 = 1513$

   (d) $z_{\text{low}} = 31 \cdot 25 = 775$

   (e) $z_{\text{medium}} = 5472 - 1513 - 775 = 3184$

   (f) $z = 1513 \cdot 10^4 + 3184 \cdot 10^2 + 775 = 15449175$

3. Multiplying $81185216 \cdot 34327283$:

$$\begin{aligned} z_{\text{total}} &= (8118 + 5216) \cdot (3432 + 7283) \\ &= 13334 \cdot 10715 \text{ (computed below)} \\ &= 142873810 \\ z_{\text{high}} &= 8118 \cdot 3432 \text{ (computed below)} \\ &= 27860976 \\ z_{\text{low}} &= 7283 \cdot 5216 \text{ (computed below)} \\ &= 37988128 \end{aligned}$$

Multiplying $13334 \cdot 10715$:

$$\begin{aligned} z_{\text{total}} &= (13 + 334) \cdot (10 + 715) \text{ (computed in hardware)} \\ &= 251575 \\ z_{\text{high}} &= 13 \cdot 10 \text{ (computed in hardware)} \\ &= 130 \\ z_{\text{low}} &= 334 \cdot 715 \text{ (computed in hardware)} \\ &= 238810 \end{aligned}$$

Multiplying $8118 \cdot 3432$:

$$\begin{aligned} z_{\text{total}} &= (81 + 18) \cdot (34 + 32) \text{ (computed in hardware)} \\ &= 6534 \\ z_{\text{high}} &= 81 \cdot 34 \text{ (computed in hardware)} \\ &= 2754 \\ z_{\text{low}} &= 18 \cdot 32 \text{ (computed in hardware)} \\ &= 576 \end{aligned}$$

Multiplying $7283 \cdot 5216$:

$$\begin{aligned} z_{\text{total}} &= (72 + 83) \cdot (52 + 16) \text{ (computed in hardware)} \\ &= 10540 \\ z_{\text{high}} &= 72 \cdot 52 \text{ (computed in hardware)} \\ &= 3744 \\ z_{\text{low}} &= 83 \cdot 16 \text{ (computed in hardware)} \\ &= 1328 \end{aligned}$$

4. $z_{\text{total}} = (8118 + 5216) \cdot (3432 + 7283) = 13334 \cdot 10715 = 142873810$. $z_{\text{high}} = 8118 \cdot 3432 = 27860976$. $z_{\text{low}} = 7283 \cdot 5216 = 37988128$. $z_{\text{medium}} = z_{\text{total}} - z_{\text{high}} - z_{\text{low}} = 142873810 - 27860976 - 37988128 = 77024706$. $z = 27860976 \cdot 10^8 + 77024706 \cdot 10^4 + 37988128 = 2786867885048128$.

Chapter 12

Strassen Matrix Multiplication

Matrix multiplication is ubiquitous in computing, and large matrix multiplications are used in tasks such as meteorology, image analysis, and machine learning. The product of two $n \times n$ matrices A and B is defined as $C = A \cdot B$, where $C_{i,j} = \sum_k A_{i,k} \cdot B_{k,j}$. Matrix multiplication can be implemented in a naive manner $\in O(n^3)$ by using three nested `for` loops, one for i, one for j, and one for k. This cubic runtime will be prohibitive for large problems, and will limit the applicability of matrix multiplication.

12.1 A recursive view of matrix multiplication

For this reason, a matrix multiplication algorithm $\in o(n^3)$ would be a significant achievement. We will proceed in a manner reminiscent of Gauß and Karatsuba multiplication. For this reason, we will first construct a naive recursive form for matrix implementation. This can be done by sub-dividing the A and B matrices into 2×2 sub-matrices, each of size $\frac{n}{2} \times \frac{n}{2}$:

$$\begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \cdot \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}.$$

Here we can see

$$\begin{aligned} C_{1,1} &= A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} \\ C_{1,2} &= A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2} \\ C_{2,1} &= A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \\ C_{2,2} &= A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}. \end{aligned}$$

An implementation is shown in Listing 12.1. The runtime of this divide and conquer will be given by the recurrence $r(n) = 8r\left(\frac{n}{2}\right) + \Theta(n^2)$, where each $n \times n$ matrix multiplication reduces to 8 multiplications of size $\frac{n}{2} \times \frac{n}{2}$ and the $\Theta(n^2)$ term is given from the element-wise sums of matrices. Using the master theorem, we observe that the number of leaves is $\ell = n^{\log_2(8)} = n^3$, which has a polynomial power significantly larger than the $\Theta(n^2)$ term, and is thus a leaf-heavy case; therefore, the runtime will be $r(n) \in \Theta(n^3)$, matching the naive non-recursive algorithm described at the start of this chapter.

Listing 12.1: Naive recursive matrix multiplication.

```
import numpy

# defined for square matrices:
# runtime is r(n) = 8*r(n/2) + n^2 \in \Theta(n^3)
def multiply_matrices(A,B,LEAF_SIZE=512):
    rA,cA=A.shape
    rB,cB=B.shape
    n = rA
    assert(n==cA and n==rB and n==cB)
    if n <= LEAF_SIZE:
        return A*B
    a11=A[:n/2,:n/2]
    a12=A[:n/2,n/2:]
    a21=A[n/2:,:n/2]
    a22=A[n/2:,n/2:]
    b11=B[:n/2,:n/2]
    b12=B[:n/2,n/2:]
    b21=B[n/2:,:n/2]
    b22=B[n/2:,n/2:]
    result = numpy.matrix(numpy.zeros( (n,n) ))
    result[:n/2,:n/2] = a11*b11 + a12*b21
    result[:n/2,n/2:] = a11*b12 + a12*b22
    result[n/2:,:n/2] = a21*b11 + a22*b21
    result[n/2:,n/2:] = a21*b12 + a22*b22
    return result
```

```
n = 1024
A = numpy.matrix(numpy.random.uniform(0.0, 1.0, n*n).reshape(n,n))
B = numpy.matrix(numpy.random.uniform(0.0, 1.0, n*n).reshape(n,n))

print 'Naive iterative (with numpy)'
exact = A*B
print exact
print

print 'Recursive'
fast = multiply_matrices(A,B)
print fast
print

print 'Absolute error', max( numpy.fabs(numpy.array(fast -
    exact).flatten()) )
```

12.2 Faster matrix multiplication

We will now begin summing sub-matrices before multiplying them (in a manner reminiscent of Karatsuba's algorithm). We do not have a concrete strategy for how to proceed, only that we should imitate our strategy in Gauß' method and Karatsuba's method. Note that here there is one additional constraint: matrix multiplication is not commutative in the general case (*i.e.*, $A \cdot B \neq B \cdot A$ on all matrices).

We follow the lead left by Karatsuba: We start by summing rows and columns before multiplying. Specifically, we'll sum rows of A and columns of B.

$$\begin{aligned} X_1 &= A_{1,1} + A_{1,2} \\ X_2 &= A_{2,1} + A_{2,2} \\ Y_1 &= B_{1,1} + B_{2,1} \\ Y_2 &= B_{1,2} + B_{2,2}. \end{aligned}$$

We can compute some products of these row and column sums as we did in Karatsuba's method. Specifically, $C_{1,2}$ contains terms composed of X_1 and

Y_2:

$$\begin{aligned} P_1 &= X_1 \cdot B_{2,2} \\ P_2 &= A_{1,1} \cdot Y_2. \end{aligned}$$

The sum

$$\begin{aligned} P_1 + P_2 &= A_{1,1} \cdot B_{2,2} + A_{1,2} \cdot B_{2,2} \ + \ A_{1,1} \cdot B_{1,2} + A_{1,1} \cdot B_{2,2} \\ &= A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2} \ + \ 2A_{1,1} \cdot B_{2,2} \\ &= C_{1,2} + 2A_{1,1} \cdot B_{2,2}. \end{aligned}$$

Notice that we have overcounted $A_{1,1} \cdot B_{2,2}$ rather than canceling it. We proceed as we did with Gauß multiplication: if we change the signs on the column sums Y_1 and Y_2, we can eliminate the $A_{1,1} \cdot B_{2,2}$ terms.

$$\begin{aligned} X_1 &= A_{1,1} + A_{1,2} \\ Y_2 &= B_{1,2} - B_{2,2} \\ P_1 &= X_1 \cdot B_{2,2} \\ P_2 &= A_{1,1} \cdot Y_2 \\ P_1 + P_2 &= A_{1,1} \cdot B_{2,2} + A_{1,2} \cdot B_{2,2} \ + \ A_{1,1} \cdot B_{1,2} - A_{1,1} \cdot B_{2,2} \\ &= A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2} \\ &= C_{1,2} \end{aligned}$$

In the same manner, we can recover $C_{2,1}$:

$$\begin{aligned} X_2 &= A_{2,1} + A_{2,2} \\ Y_1 &= B_{2,1} - B_{1,1} \\ P_3 &= X_2 \cdot B_{1,1} \\ P_4 &= A_{2,2} \cdot Y_1 \\ P_3 + P_4 &= A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{1,1} \ + \ A_{2,2} \cdot B_{2,1} - A_{2,2} \cdot B_{1,1} \\ &= A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \\ &= C_{2,1}. \end{aligned}$$

We have already incurred 4 sub-matrix multiplications (remember that the naive method uses 8 sub-matrix multiplications total); therefore, we will now try to find $C_{1,1}$ and $C_{2,2}$ by using ≤ 3 more sub-matrix multiplications.

Remember that $C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$. We have not yet combined $A_{1,1} \cdot B_{1,1}$ nor $A_{1,2} \cdot B_{2,1}$. Likewise, remember that $C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$, and we have not yet combined $A_{2,1} \cdot B_{1,2}$ nor $A_{2,2} \cdot B_{2,2}$. To combine $A_{1,1} \cdot B_{1,1}$ and $A_{2,2} \cdot B_{2,2}$, in the same product, we will compute

$$P_5 = (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2}) .$$

P_5 does not yet compute $A_{1,2} \cdot B_{2,1}$ (needed by $C_{1,1}$) and does not yet compute $A_{2,1} \cdot B_{1,2}$ (needed by $C_{2,2}$). Also, P_5 includes terms $A_{1,1} \cdot B_{2,2} + A_{2,2} \cdot B_{1,1}$, which are unwanted by both $C_{1,1}$ and $C_{2,2}$.

Fortunately, our previously computed P_2 contributes a $-A_{1,1} \cdot B_{2,2}$ term and P_4 contributes a $-A_{2,2} \cdot B_{1,1}$ term:

$$P_5 + P_2 + P_4 = A_{1,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,2} + A_{1,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,1}.$$

$$\begin{aligned}
C_{1,1} - (P_5 + P_2 + P_4) &= A_{1,2} \cdot B_{2,1} - (A_{2,2} \cdot B_{2,2} + A_{1,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,1}) \\
&= (A_{1,2} - A_{2,2}) \cdot B_{2,1} - A_{2,2} \cdot B_{2,2} - A_{1,1} \cdot B_{1,2} \\
&= (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2}) - A_{1,2} \cdot B_{2,2} - A_{1,1} \cdot B_{1,2} \\
C_{2,2} - (P_5 + P_2 + P_4) &= A_{2,1} \cdot B_{1,2} - (A_{1,1} \cdot B_{1,1} + A_{1,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,1}) \\
&= (A_{2,1} - A_{1,1}) \cdot B_{1,2} - A_{1,1} \cdot B_{1,1} - A_{2,2} \cdot B_{2,1} \\
&= (A_{2,1} - A_{1,1}) \cdot (B_{1,1} + B_{1,2}) - A_{2,1} \cdot B_{1,1} - A_{2,2} \cdot B_{2,1}
\end{aligned}$$

The last of these steps in each of the above equations is perhaps the most mysterious. We do this because we have a $-A_{2,2} \cdot B_{2,2}$ term in the $C_{1,1}$ equation and a $-A_{1,1} \cdot B_{1,1}$ term in the $C_{2,2}$ equation. Consider that the entire point of constructing P_5 as we did was to compute the products $A_{1,1} \cdot B_{1,1}$ and $A_{2,2} \cdot B_{2,2}$ (because they have not yet occurred in any P_1, P_2, P_3, or P_4); therefore, any step where we still need to compute those values would essentially be circular. For this reason, the last step in each of the equations above factors to exchange the $-A_{2,2} \cdot B_{2,2}$ and $-A_{1,1} \cdot B_{1,1}$ terms for $-A_{1,2} \cdot B_{2,2}$ and $-A_{2,1} \cdot B_{1,1}$, respectively.

The key to moving forward from there is seeing that the residual $A_{1,2} \cdot B_{2,2} + A_{1,1} \cdot B_{1,2}$ includes terms from both P_2 and P_1. P_1 came from the aggregate first row of A combined with $B_{2,2}$ and P_2 came from $A_{1,1}$ with the aggregate second column of B: the term where P_1 and P_2 overlap is the precise term that P_1 and P_2 contain but the residual $A_{1,2} \cdot B_{2,2} + A_{1,1} \cdot B_{1,2}$

does not. The same is true for the second residual $A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$; it matches terms from P_4 and P_3, and once again the term they share is the term we wish to delete.

For this reason, the residuals $A_{1,2} \cdot B_{2,2} + A_{1,1} \cdot B_{1,2}$ and $A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$ can be reconstructed using products already computed. We can find this by searching our already computed products for parts of these residuals. From this, we can see the following:

$$\begin{aligned} A_{1,2} \cdot B_{2,2} + A_{1,1} \cdot B_{1,2} &= P_2 + P_1 \\ &= A_{1,1} \cdot (B_{1,2} - B_{2,2}) + (A_{1,1} + A_{1,2}) \cdot B_{2,2} \\ A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} &= P_4 + P_3 \\ &= A_{2,2} \cdot (B_{2,1} - B_{1,1}) + (A_{2,1} + A_{2,2}) \cdot B_{1,1}. \end{aligned}$$

As we aimed for, we have only used 7 products (with $P_6 = (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2})$ and $P_7 = (A_{2,1} - A_{1,1}) \cdot (B_{1,1} + B_{1,2})$, which were computed above):

$$\begin{aligned} C_{1,1} &= P_5 + P_4 - P_1 + P_6 \\ C_{2,2} &= P_5 + P_2 - P_3 + P_7. \end{aligned}$$

Thus, we can reduce an $n \times n$ matrix product to 7 $\frac{n}{2} \times \frac{n}{2}$ matrix products and a constant number of matrix additions and subtractions. Thus we have $r(n) = 7r\left(\frac{n}{2}\right) + \Theta(n^2)$. Using the master theorem, this is a leaf-heavy case with runtime $\in \Theta(n^{\log_2(7)}) \approx \Theta(n^{2.807\ldots})$.

An implementation of Strassen's algorithm is shown in Listing 12.2.

Listing 12.2: Strassen's fast matrix multiplication algorithm.

```
import numpy

# defined for square matrices:
def strassen(A,B,LEAF_SIZE=512):
    rA,cA=A.shape
    rB,cB=B.shape
    n = rA
    assert(n==cA and n==rB and n==cB)
    if n <= LEAF_SIZE:
        return A*B
    a11=A[:n/2,:n/2]
    a12=A[:n/2,n/2:]
```

```
    a21=A[n/2:,:n/2]
    a22=A[n/2:,n/2:]
    b11=B[:n/2,:n/2]
    b12=B[:n/2,n/2:]
    b21=B[n/2:,:n/2]
    b22=B[n/2:,n/2:]
    p1=strassen((a11+a12),b22)
    p2=strassen(a11,(b12-b22))
    p3=strassen((a21+a22),b11)
    p4=strassen(a22,(b21-b11))
    p5=strassen((a11+a22),(b11+b22))
    p6=strassen((a12-a22),(b21+b22))
    p7=strassen((a21-a11),(b11+b12))
    result = numpy.matrix(numpy.zeros( (n,n) ))
    result[:n/2,:n/2] = p5 + p4 - p1 + p6
    result[:n/2,n/2:] = p2 + p1
    result[n/2:,:n/2] = p3 + p4
    result[n/2:,n/2:] = p5 + p2 - p3 + p7
    return result

n = 1024
A = numpy.matrix(numpy.random.uniform(0.0, 1.0, n*n).reshape(n,n))
B = numpy.matrix(numpy.random.uniform(0.0, 1.0, n*n).reshape(n,n))

print 'Naive'
exact = A*B
print exact
print

print 'Strassen'
fast = strassen(A,B,4)
print fast
print

print 'Absolute error', max( numpy.fabs(numpy.array(fast -
    exact).flatten()) )
```

12.3 Zero padding

When n is not a power of 2, we can simply round n up to the next power of 2 and "pad" with zeros so that our matrix of interest is embedded into a larger square matrix whose width is a power of 2. We could then multiply

this larger matrix with Strassen's algorithm. The runtime of this approach will be no more than $r(2n)$, which is still $\in O(n^{\log_2(7)})$.

In practice, it is generally practically faster to change to a naive algorithm when n becomes small enough. That approach is more flexible, because including base cases such as 3×3 matrices means that the divide-and-conquer method can include a factor of 3 as well as powers of 2. This approach can be paired with zero padding.

12.4 The search for faster algorithms

To date, the fastest known matrix multiplication algorithms use a more complicated variant of Strassen's approach. Those algorithms have exponents ≈ 2.373, and faster algorithms have not yet been discovered. Likewise, it is not yet known whether faster algorithms are *possible*[1].

These fancier, asymptotically faster algorithms are rarely used in practice, because only problems with a large n will allow the faster asymptotic complexity to overcome the larger runtime constant of those fancier algorithms; however, the simpler Strassen algorithm and its variants are used in high-performance linear algebra libraries.

Matrix multiplication must be $\in \Omega(n^2)$, because we at least need to load the two matrix arguments and to eventually write the matrix result. If we ever discover matrix multiplication algorithms $\in O(n^{2+\epsilon})$ (note that this would include runtimes such as $n^2(\log(n))^{1000}$), that discovery will have important implications on other important problems. For that reason, people are actively searching for improvements.

[1] The distinction between discovery and proven existence is exemplified by saying that even though I have not, at the time of writing, seen the film "Lethal Weapon 4", I am assured by the internet that it does exists. With faster matrix multiplication algorithms, we do not even have that knowledge of existence. Likewise, we have not yet proven that no such method can exist (as we proved that sorting using only a comparison operator cannot possibly run with worst-case runtime $\in o(n \log(n))$).

Chapter 13

Fast Polynomial Multiplication and The Fast Fourier Transform

Consider the problem of multiplying two polynomials, $a(x)$ and $b(x)$:

$$\begin{aligned} a(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \\ b(x) &= b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1} \\ c(x) &= a(x) \cdot b(x) \\ &= c_0 + c_1 x + c_2 x^2 + \cdots + c_{2n-2} x^{2n-2}. \end{aligned}$$

We can find, c_m, an arbitrary coefficient of $c(x)$, by summing all possible contributions from $a(x)$ and $b(x)$:

$$\begin{aligned} c_m &= \sum_{i,k:i+k=m} a_i \cdot b_k \\ &= \sum_i a_i \cdot b_{m-i}. \end{aligned}$$

Given two vectors, a and b, with the coefficients of two polynomials $a(x)$ and $b(x)$, finding the c vector, the coefficients of $c(x)$, is called the "convolution" of the a and b vectors and is denoted as follows:

$$c = a \circledast b.$$

Convolution is not only important for directly related mathematical tasks (*e.g.*, as multiplying polynomials), it is also highly important in smoothing images, in signal processing, and dynamic programming algorithms.

13.1 Naive convolution

Naive convolution directly uses the equation defining c_m. The runtime of this is clearly the number of m visited times the number of terms that must be summed for each of those m:

$$\begin{aligned} s(n) &= \sum_{m=0}^{2n-1} |\{(i,k) : i+k=m,\ i \in \{0,1,\ldots n-1\}\ k \in \{0,1,\ldots n-1\}\ \}| \\ &= \sum_{m=0}^{2n-1} m \\ &\in \Theta(n^2). \end{aligned}$$

The runtime of the naive algorithm is quadratic in the polynomial size.

At first it may seem that there is no way to signifcantly improve over naive convolution: after all, each element a_i touches each element b_k; however, this was the same flavor of reasoning that led Kolmogorov astray in Chapter 11. To build a faster algorithm, we should not think about the process and the steps performed, but instead think about the *meaning* of multiplying polynomials.

13.2 Defining polynomials by the points they pass through

To start with, let us consider how we can define a polynomial. One method is using the coefficient vector as above. But another definition is using the *points* that the polynomial passes through. If we give n unique x values, $x_0, x_1, \ldots x_{n-1}$, then knowing $a(x_0), a(x_1), \ldots a(x_{n-1})$ is sufficient to uniquely determine the polynomial[1].

We can prove that n points uniquely determine a polynomial with n coefficients. Given n points through which polynomial a passes, $(x_0, a(x_0)), (x_1, a(x_1)), \ldots (x_{n-1}, a(x_{n-1}))$, assume that some other polynomial distinct from a would pass through those same points: $t(x_0) = a(x_0), t(x_1) = a(x_1), \ldots$. Let the polynomial $s(x) = a(x) - t(x)$. Like a and t, $s(x)$ has n coefficents. Clearly, $s(x_0) = 0, s(x_1) = 0, \ldots$ because

[1] If this is not immediately intuitive, consider that you need two points to uniquely fit a line, three points to uniquely fit a quadratic, four points to uniquely fit a cubic, *etc.*

$a(x_0) = t(x_0), a(x_1) = t(x_1), \ldots$. For this reason, $x_0, x_1, \ldots$ constitute n unique zeros of the polynomial $s(x)$. We can therefore write $s(n)$ in its factored form

$$s(n) = z \cdot (x - x_0) \cdot (x - x_1) \cdot (x - x_2) \cdots (x - x_{n-1}).$$

Note that z is a constant that allows us to scale polynomials with identical zeros. We can expand this polynomial to have a term of the form zx^n. Since $a(x)$ and $t(x)$ have no x^n terms (their highest terms have power $n - 1$), then z must be 0. Thus, we can see that $s(x) = 0$, which is the same as saying $t(x) = a(x)$. This proves by contradiction that having n unique points through which a passes will uniquely determine $a(x)$ and its coefficient vector a.

13.3 Multiplying polynomials using their interpolated forms

First, let's assume that we can compute n through which $a(x)$ passes in $o(n^2)$. Then, we could do the same for $b(x)$ using the same points $x_0, x_1, \ldots x_{n-1}$. From this, in $O(n)$ we can directly compute the points through which $c(x)$ passes:

$$\forall i,\ c(x_i) = a(x_i) \cdot b(x_i).$$

Then, if we have a method running $\in o(n^2)$ that will convert back from points through which $c(x)$ passes to the coefficients c, we will have solved the convolution in $o(n^2)$.

13.4 Fast evaluation at points

It turns out that our most significant task will be computing $a(x_0), a(x_1), \ldots$ from the coefficient vector a. In the naive case, each polynomial evaluation is $\in O(n)$, because it sums n terms[2]; therefore, naively evaluating $a(x)$ at n unique points $x_0, x_1, \ldots x_{n-1}$ will cost $\in \Omega(n^2)$.

[2]Computing the powers of x when a function like `numpy.power` is not available can be done by summing the lower-power terms first and using the recurrence $x^n = x^{n-1} \cdot x$ to compute the next power of x in $O(1)$ time.

It may seem we are stuck, but remember that proving that one particular solution to a problem is slow does not yet prove that there is no fast solution. Here, we will concern ourselves with finding ways that, given $a(x_i)$ we can get $a(x_k)$ more efficiently than in the naive case. A classic case of this is when $a(x)$ is an even polynomial and where $x_k = -x_i$:

$$\begin{aligned} a(x) &= a_0 + 0 + a_2x^2 + 0 + a_4x^4 + 0 + \cdots \\ a(x_k) &= a_0 + 0 + a_2(-x_i)^2 + 0 + a_4(-x_i)^4 + 0 + \cdots \\ &= a_0 + 0 + a_2x_i^2 + 0 + a_4x_i^4 + 0 + \cdots \\ &= a(x_i). \end{aligned}$$

A similar property exists when $a(x)$ is an odd polynomial, and we get $a(-x) = -a(x)$ (thus $a(x_k) = -a(x_i)$). In this manner, if we have either even or odd polynomials and choose our $x_0, x_1, \ldots$ strategically (so that the first half x_i are the negatives of the second half of the $x_{\frac{n}{2}+i}$), then we may find a shortcut around the quadratic runtime.

Of course, in general $a(x)$ is not going to be purely an even polynomial, nor is it going to be a purely odd polynomial. But *any* polynomial can be partitioned into its even and odd parts:

$$\begin{aligned} a(x) &= a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + \cdots \\ a(x) &= a_{even}(x) + a_{odd}(x) \\ a_{even}(x) &= a_0 + a_2x^2 + a_4x^4 + \cdots \\ a_{odd}(x) &= a_1x + a_3x^3 + a_5x^5 + \cdots . \end{aligned}$$

In this manner, if we strategically choose half of our x_i values to be the negatives of the first half, then we can try to use this to more efficiently evaluate $a(x)$ at all points. Once we split $a(x)$ into $a_{even}(x)$ and $a_{odd}(x)$, then we get

$$\begin{aligned} a_{even}(x_{\frac{n}{2}+i}) &= a_{even}(x_i) \\ a_{odd}(x_{\frac{n}{2}+i}) &= -a_{odd}(x_i). \end{aligned}$$

From this we can compute $a(x) = a_{even}(x) + a_{odd}(x)$ at every point x_i, but by computing only half of the $a(x_i)$. Furthermore, we have reduced polynomial multiplication to smaller polynomial multiplications. For this reason, it may feel like we are close to being finished with constructing a divide-and-conquer algorithm, but this is not yet the case.

13.4.1 Packing polynomial coefficients

First, there is a caveat to this approach: In each recursion, we store a coefficient vector of length n. For example in the even recursion, we send coefficients $a_{even} = [a_0, 0, a_2, 0, a_4, 0, \ldots]$. Just *constructing* these coefficient vectors in every recursion will correspond to runtime recurrence $r(n) = 2r(n)+n$, because the coefficient vectors do not shrink. For this reason, the runtime will be n in the first call, n in its recursive call to evaluate $a_{even}(x)$ and n it its recursive call to evaluate $a_{odd}(x)$, and so on. The master theorem cannot be applied to problems that do not shrink, and for good reason: at every level of the tree, the runtime will actually *grow* rather than break down as we would like in a divide-and-conquer algorithm.

For this reason, we need to "pack" the coefficient vectors down into a dense form that excludes the 0 values. For example,

$$pack([a_0, 0, a_2, 0, a_4, 0, a_6, \ldots]) = [a_0, a_2, a_4, a_6, \ldots].$$

When we recursively evaluate $a_{even}(x)$, we would like to do so using the packed vector of the even-power coefficients; however, if we directly recurse with $pack(a_{even}) = [a_0, a_2, a_4, a_6, \ldots]$, we will be evaluating polynomial $a_0 + a_2x+a_4x^2+a_6x^3+\cdots$ at our chosen points rather than evaluating polynomial $a_0 + a_2x^2 + a_4x^4 + a_6x^6 + \cdots = a_{even}(x)$. This can be resolved by replacing every x with x^2 in our evaluation:

$$\begin{aligned} pack(a_{even})(x) &= a_0 + a_2x + a_4x^2 + a_6x^3 + \cdots \\ pack(a_{even})(x^2) &= a_0 + a_2x^2 + a_4x^4 + a_6x^6 + \cdots \\ &= a_{even}(x). \end{aligned}$$

13.4.2 Converting odd polynomials to even polynomials

When doing the same to compute $a_{odd}(x)$, there is an additional subtlety: we do not want to write a different polynomial evaluation function for even and odd polynomials (ideally, we would have one unified function). Here, we can use the fact that $a_{odd}(x)$ can be written as the product of x and another even polynomial, $u(x)$: $a_{odd}(x) = x \cdot u(x)$, where $u(x) = a_1 + a_3x^2 + a_5x^4 + \cdots$ and equivalently its coefficient vector is $u = [a_1, 0, a_3, 0, a_5, \ldots]$. Thus we see

that

$$\begin{aligned} a(x) &= a_{even}(x) + a_{odd}(x) \\ &= a_{even}(x) + x \cdot u(x) \\ &= pack(a_{even})(x^2) + x \cdot pack(u)(x^2). \end{aligned}$$

Thus we have reduced polynomial evaluation to two *smaller* polynomial evaluations (smaller in both coefficient vector length and in number of points on which they are evaluated).

13.4.3 Complex roots of unity

As above, we choose our unique $x_0, x_1, ...$ values strategically so that the first half are the negatives of the second half in every recursion: $\forall i < \frac{n}{2},\ x_i = -x_{\frac{n}{2}+i}$. In this manner, we find that

$$\begin{aligned} a(x_i) &= pack(a_{even})(x_i^2) + x_i \cdot pack(u)(x_i^2) \\ a(x_{\frac{n}{2}+i}) &= pack(a_{even})(x_{\frac{n}{2}+i}^2) + x_{\frac{n}{2}+i} \cdot pack(u)(x_{\frac{n}{2}+i}^2) \\ &= pack(a_{even})(x_i^2) + x_{\frac{n}{2}+i} \cdot pack(u)(x_i^2) \\ &= pack(a_{even})(x_i^2) - x_i \cdot pack(u)(x_i^2), \end{aligned}$$

because $x_i = -x_{\frac{n}{2}+i}$ means that $x_i^2 = (-x_{\frac{n}{2}+i})^2 = x_{\frac{n}{2}+i}^2$. This is the magic step to constructing our divide-and-conquer algorithm: we only recursively evaluate the polynomials at the first half of the points $\forall i < \frac{n}{2},\ x_i$, and get the evaluations on the other half of the points ($\forall i < \frac{n}{2},\ x_{\frac{n}{2}+i}$) for free. In this manner, we've reduced an evaluation of a polynomial with n coefficients at n points to two evaluations of polynomials with $\frac{n}{2}$ coefficients at $\frac{n}{2}$ points. Thus, we have a runtime recurrence $r(n) = 2r\left(\frac{n}{2}\right) + n$.

However, to continue the recursions in this manner, we need to enforce our invariant that the first half of the points evaluated in that recursion will be the negatives of the second half of the points evaluated. In the first recursion, this is simply $\forall i < \frac{n}{2},\ x_i = -x_{\frac{n}{2}+i}$. But in the second recursion, we have already replaced x with x^2 (when calling the first recursion), and so we need to enforce $\forall i < \frac{n}{4},\ x_i^2 = -x_{\frac{n}{4}+i}^2$. That is, the first half of the points on which we evaluate should be the negative of the second half of the points. But the first quarter of the points should have values that, when squared are the negatives of the second quarter squared.

It may initially seem imposible for $x_0^2 = x_{\frac{n}{4}}^2$; with standard real numbers, it is impossible. But not if we use complex numbers: $x_0 = x_{\frac{n}{4}} \cdot j$ will suffice (where $j = \sqrt{-1}$ following the notation in Chapter 11).

In the next recursion, we will have $\forall i < \frac{n}{8},\ x_i^4 = -x_{\frac{n}{8}+i}^4$. If we already know that our x_i points must be complex numbers, then it is advantageous to think in terms of the polar form $x_i = e^{\theta_i \cdot j}$, because this can be taken to powers easily:

$$x_i^k = e^{\theta_i \cdot j^k} = e^{\theta_i \cdot k \cdot j}.$$

In this manner, we see that $\forall i < \frac{n}{8},\ x_i^4 = -x_{\frac{n}{8}+i}^4$ is equivalent to saying

$$\forall i < \frac{n}{8},\ e^{\theta_i 4j} = -e^{\theta_{\frac{n}{4}+i} 4j}.$$

We know that $e^{\theta \cdot j} = \cos(\theta) + j \cdot \sin(\theta)$; therefore, for two angles θ_x and θ_y, $e^{\theta_x \cdot k \cdot j} = -e^{\theta_y \cdot k \cdot j}$ requires

$$\cos(\theta_x \cdot k) + j \cdot \sin(\theta_x \cdot k) = -\left(\cos(\theta_y \cdot k) + j \cdot \sin(\theta_y \cdot k)\right),$$

which requires both of the following:

$$\begin{aligned} \cos(\theta_x \cdot k) &= -\cos(\theta_y \cdot k) \\ \sin(\theta_x \cdot k) &= -\sin(\theta_y \cdot k). \end{aligned}$$

Both of these requirements are satisfied simultaneously by $\theta_y = \theta_x + \frac{\pi}{k}$:

$$\begin{aligned} \cos(\theta_x \cdot k) &= -\cos((\theta_x + \frac{\pi}{k}) \cdot k) \\ &= -\cos(\theta_x \cdot k + \pi) \\ &= \cos(\theta_x \cdot k) \\ \sin(\theta_x \cdot k) &= -\sin((\theta_x + \frac{\pi}{k}) \cdot k) \\ &= -\sin(\theta_x \cdot k + \pi) \\ &= \sin(\theta_x \cdot k). \end{aligned}$$

For this reason, we can see that each point in the second half of our points should have angle $+\pi$ from some point in the first half of the points. Likewise, each point in the second quarter of our points should have angle $+\frac{\pi}{2}$ from some point in the second quarter of the points. Similarly, each point in the second eighth of our points should have some angle $+\frac{\pi}{4}$ from some point in

the first eighth of our points. Continuing in this manner, we will eventually narrow down to only two points, where one has angle $+\frac{\pi}{2^k}$ from the other where k is the number of times we needed to divide the points in half before reaching 2. $k = \log_2(n) - 1$ and this difference between the angles will be

$$\frac{\pi}{2^{\log_2(n)-1}} = \frac{\pi}{n/2} = \frac{2\pi}{n}.$$

Thus, if $\theta_0 = 0$, then $\theta_1 = \frac{2\pi}{n}$. Expanding upwards we see that $\theta_2 = \theta_0 + \frac{\pi}{n} = \frac{\pi}{n}$ and $\theta_3 = \theta_1 + \frac{\pi}{n} = \frac{2\pi}{n} + \frac{\pi}{n} = \frac{3\pi}{n}$. Continuing on in this manner, we see that $\theta_i = \frac{2\pi \cdot i}{n}$. Thus, $x_i = e^{\frac{2\pi \cdot i}{n} \cdot j}$. These values are sometimes referred to as "the complex roots of unity" because they are n non-redundant complex solutions to the equation $x^n = 1$.

By choosing these points on which to evaluate our polynomial, we complete the invariant that at every recursion the first half of our points will be the negative of the second half (including the packing step, on which we replaced x with x^2).

13.4.4 Runtime of our algorithm

Listing 13.1 shows our resulting divide-and-conquer algorithm, which has runtime $r(n) = 2r\left(\frac{n}{2}\right) + n$. The master theorem reveals this to be root-leaf balanced, which has closed form $r(n) \in \Theta(n \log(n))$. This is a substantial improvement over the naive $\Theta(n^2)$ approach to naively compute n points through which our polynomial passes. This is an algorithm discovered by Cooley and Tukey. Because of the relationship between complex exponentials and trigonometric functions, this approach is named "the fast Fourier" transform, after namesake Joseph Fourier, who pioneered the notion of writing arbitrary patterns in terms of trigonometric functions. For these reasons, we would say we've implemented the "Cooley-Tukey FFT".

Listing 13.1: The Cooley-Tukey FFT, a $\Theta(n \log(n))$ algorithm to evaluate a polynomial at n distinct points.

```
import numpy
from time import time

# this is an r(n) = 2*r(n/2) + \Theta(n) \in \Theta(n log(n)) algorithm:
def fft(vec):
  n=len(vec)
```

```
  if n==1:
    return vec

  result = numpy.array(numpy.zeros(n), numpy.complex128)

  # packed coefficients eliminate zeros. e.g., f(x)=1+2x+3x**2+...,
  # then e(x)=1+3x**2+... = 1+0x+3x**2+0x**3+... = (1+3y+...),y=x**2.
  packed_evens = vec[::2]
  packed_odds = vec[1::2]

  # packed_evens(x**2) and packed_odds(x**2) for the first half of x
  # points. The other half of the points are the negatives of the
  # first half (used below).
  fft_evens = fft(packed_evens)
  fft_odds = fft(packed_odds)

  # Butterfly:

  for i in xrange(n/2):
    # result = evens(x) + x*odds(x), where x is a complex root of unity
    #        = packed_evens(x**2) + x*packed_odds(x**2)
    x = numpy.exp(-2*numpy.pi*i*1j/n)
    result[i] = fft_evens[i] + x * fft_odds[i]

  for i in xrange(n/2,n):
    # result = evens(x) + x*odds(x), where x is a complex root of unity
    #        = packed_evens(x**2) + x*packed_odds(x**2)
    x=numpy.exp(-2*numpy.pi*i*1j/n)
    # first half of points are negative of second half.
    # x_i = -x_{i+n/2}, x_i**2 = x_{i+n/2}**2; therefore
    # packed_evens(x_i**2) = packed_evens(x_{i+n/2}**2) and
    # packed_odds(x_i**2) = packed_odds(x_{i+n/2}**2)
    result[i] = fft_evens[i - n/2] + x * fft_odds[i - n/2]

  return result

if __name__=='__main__':
  N=2**15
  x=numpy.array(numpy.arange(N),float)

  t1=time()
  numpy_result = numpy.fft.fft(x)
  t2=time()
  print 'numpy fft:', numpy_result
```

```
print 'took', t2-t1, 'seconds'

print
t1=time()
recursive_result = fft(x)
t2=time()
print 'fast ft:', recursive_result
print 'took', t2-t1, 'seconds'

print
print 'Largest error', max(numpy.abs(numpy_result - recursive_result))
print 'Recursive python FFT took', t2-t1, 'seconds'
```

13.5 In-place FFT computation

Currently, our FFT makes allocations in each recursion (*e.g.*, when computing and storing $a_{even}(x_0), a_{even}(x_1), a_{even}(x_2), \ldots$). We can construct an in-place FFT as described in Chapter 15 of of "Code Optimization in `C++11`" (Serang 2018) by performing a bit-reversed permutation in advance.

13.6 Going from points to coefficients

We have already accomplished our first task of going from coefficients to points in $o(n^2)$ time. Using this approach, we can now get $c(x_0), c(x_1), \ldots$ in $O(n \log(n) + n) = O(n \log(n))$ time. Now all that remains is to convert $c(x)$ back from points through which it passes into its coefficient vector $c = [c_0, c_1, c_2, \ldots]$.

The point $x_1 = e^{\frac{2\pi \cdot i}{n} \cdot j}$ is the point with the smallest nonzero angle. With this point, we can generate any x_i we like: $x_i = x_1^i$; therefore, can write our

FFT in a naive manner:

$$\begin{bmatrix} a(x_0) \\ a(x_1) \\ a(x_2) \\ \vdots \\ a(x_{n-1}) \end{bmatrix} = \begin{bmatrix} x_0^0 & x_0^1 & x_0^2 & \cdots & x_0^{n-1} \\ x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^{n-1} \\ x_2^0 & x_2^1 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & & & & \\ x_{n-1}^0 & x_{n-1}^1 & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} . \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

$$= \begin{bmatrix} x_1^0 & x_1^0 & x_1^0 & \cdots & x_1^0 \\ x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^{n-1} \\ x_1^0 & x_1^2 & x_1^4 & \cdots & x_1^{2\cdot(n-1)} \\ \vdots & & & & \\ x_1^{0\cdot(n-1)} & x_1^{1\cdot(n-1)} & x_1^{2\cdot(n-1)} & \cdots & x_1^{(n-1)\cdot(n-1)} \end{bmatrix} . \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} .$$

To go back from points through which a polynomial passes to the coefficients of the polynomial, we need to solve the above equation for $[a_0, a_1, \ldots a_{n-1}]^T$. If we begin by denoting the matrix

$$G = \begin{bmatrix} x_1^0 & x_1^0 & x_1^0 & \cdots & x_1^0 \\ x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^{n-1} \\ x_1^0 & x_1^2 & x_1^4 & \cdots & x_1^{2\cdot(n-1)} \\ \vdots & & & & \\ x_1^{0\cdot(n-1)} & x_1^{1\cdot(n-1)} & x_1^{2\cdot(n-1)} & \cdots & x_1^{(n-1)\cdot(n-1)} \end{bmatrix}$$

$$= \begin{bmatrix} e^{\frac{2\pi\cdot 0\cdot 0\cdot j}{n}} & e^{\frac{2\pi\cdot 0\cdot 1\cdot j}{n}} & e^{\frac{2\pi\cdot 0\cdot 2\cdot j}{n}} & \cdots & e^{\frac{2\pi\cdot 0\cdot (n-1)\cdot j}{n}} \\ e^{\frac{2\pi\cdot 1\cdot 0\cdot j}{n}} & e^{\frac{2\pi\cdot 1\cdot 1\cdot j}{n}} & e^{\frac{2\pi\cdot 1\cdot 2\cdot j}{n}} & \cdots & e^{\frac{2\pi\cdot 1\cdot (n-1)\cdot j}{n}} \\ e^{\frac{2\pi\cdot 0\cdot 0\cdot j}{n}} & e^{\frac{2\pi\cdot 2\cdot 1\cdot j}{n}} & e^{\frac{2\pi\cdot 2\cdot 2\cdot j}{n}} & \cdots & e^{\frac{2\pi\cdot 2\cdot (n-1)\cdot j}{n}} \\ \vdots & & & & \\ e^{\frac{2\pi\cdot (n-1)\cdot 0\cdot j}{n}} & e^{\frac{2\pi\cdot (n-1)\cdot 1\cdot j}{n}} & e^{\frac{2\pi\cdot (n-1)\cdot 2\cdot j}{n}} & \cdots & e^{\frac{2\pi\cdot (n-1)\cdot (n-1)\cdot j}{n}} \end{bmatrix},$$

then we can solve the equation by inverting the matrix:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = G^{-1} . \begin{bmatrix} a(x_0) \\ a(x_1) \\ a(x_2) \\ \vdots \\ a(x_{n-1}) \end{bmatrix} .$$

Computing the actual matrix inverse will be slower than naive convolution, and so we would like to avoid that if possible.

Fortunately, it turns out that

$$H = \frac{1}{n}\begin{bmatrix} e^{-\frac{2\pi\cdot 0\cdot 0\cdot j}{n}} & e^{-\frac{2\pi\cdot 0\cdot 1\cdot j}{n}} & e^{-\frac{2\pi\cdot 0\cdot 2\cdot j}{n}} & \cdots & e^{-\frac{2\pi\cdot 0\cdot (n-1)\cdot j}{n}} \\ e^{-\frac{2\pi\cdot 1\cdot 0\cdot j}{n}} & e^{-\frac{2\pi\cdot 1\cdot 1\cdot j}{n}} & e^{-\frac{2\pi\cdot 1\cdot 2\cdot j}{n}} & \cdots & e^{-\frac{2\pi\cdot 1\cdot (n-1)\cdot j}{n}} \\ e^{-\frac{2\pi\cdot 0\cdot 0\cdot j}{n}} & e^{-\frac{2\pi\cdot 2\cdot 1\cdot j}{n}} & e^{-\frac{2\pi\cdot 2\cdot 2\cdot j}{n}} & \cdots & e^{-\frac{2\pi\cdot 2\cdot (n-1)\cdot j}{n}} \\ \vdots & & & & \\ e^{-\frac{2\pi\cdot (n-1)\cdot 0\cdot j}{n}} & e^{-\frac{2\pi\cdot (n-1)\cdot 1\cdot j}{n}} & e^{-\frac{2\pi\cdot (n-1)\cdot 2\cdot j}{n}} & \cdots & e^{-\frac{2\pi\cdot (n-1)\cdot (n-1)\cdot j}{n}} \end{bmatrix}.$$

This matrix can be built by reversing the sign of the exponents in the FFT and dividing every result by n. To discover this from scratch without introducing substantial new theory, it would be possible to use Gaussian elimination to solve the equations in $O(n^3)$ time on some small problem and then observe that $H = G^*$, the complex conjugate of G. From there, it would be possible to verify the hypothesis that this pattern is maintained by checking that have product $H \cdot G$ will be filled with 1 along the diagonal and products of 0 everywhere else:

$$(G^* \cdot G)_{i,k} = \begin{cases} 1, & i = k \\ 0, & i \neq k \end{cases}.$$

As a result, doing an inverse FFT is almost identical to doing a forward FFT and scaling the result by $\frac{1}{n}$. Alternatively, this can be done as

$$ifft(a) = \frac{fft(a^*)^*}{n},$$

meaning we conjugate each element of our input vector a, perform the FFT, conjugate the result, and then scale every element by $\frac{1}{n}$. In this manner, we do not need to implement another divide-and-conquer algorithm; instead, we've simply reduced inverse FFT to FFT.

13.7 Fast polynomial multiplication

There is but one more subtlety remaining when multiplying two polynomials. Consider that when $a(x)$ and $b(x)$ are each defined by n coefficients. Each can alternatively be defined by n points that it passes through. But $c(x) = a(x) \cdot b(x)$ will be defined by $2n - 1$ coefficients. If we are trying to do fast

polynomial multiplication as described above, we will only have n points through which $c(x)$ passes, and we will actually need $2n-1$ points. For this reason, we will evaluate both $a(x)$ and $b(x)$ at $2n$ distinct points, which we will use to compute $c(x)$ at those $2n$ points. This will be sufficient to define $c(x)$, because because $2n > 2n-1$.

In our FFT implementation, we evaluate a polynomial $a(x)$ with n coefficients at n points; this is hard-coded into our FFT, and so we cannot easily modify it to evaluate the polynomial at $2n$ points. But we *can* evaluate that polynomial at $2n$ points by adding n coefficients of value 0. For example, $a(x) = a_0 + a_1 x$ could be zero padded to $a(x) = a_0 + a_1 x + 0x^2 + 0x^3$. By "zero padding" in this manner, we compute $2n$ points through which $a(x)$ passes, $2n$ points through which $b(x)$ passes, use those to get $2n$ points through which $c(x)$ passes, and then compute c, the coefficients of $c(x)$. This convolution result is demonstrated in Listing 13.2. Thus we have implemented polynomial multiplication in $\Theta(n \log(n))$.

Listing 13.2: FFT convolution (*i.e.*, polynomial multiplication) running in $\Theta(n \log(n))$.

```
from fft import *

def naive_convolve(x,y):
  assert(len(x) == len(y))
  N = len(x)

  z=numpy.zeros(len(x)+len(y)-1)
  for i in xrange(N):
    for j in xrange(N):
      z[i+j] += x[i]*y[j]
  return z

def fft_convolve(x,y):
  assert(len(x) == len(y))
  N = len(x)

  zero_pad_x = numpy.zeros(2*N)
  zero_pad_y = numpy.zeros(2*N)

  zero_pad_x[0:N] = x
  zero_pad_y[0:N] = y

  fft_zero_pad_x = fft(zero_pad_x)
  fft_zero_pad_y = fft(zero_pad_y)
```

```
  # Multiply element-wise:
  fft_zero_pad_x *= fft_zero_pad_y

  # Conjugate element wise:
  result = numpy.conj(fft_zero_pad_x)
  result = fft(result)
  result /= float(2*N)

  return result[:2*N-1]

a=[1,5,4,2]
b=[7,-2,5,1]
print 'naive convolve', naive_convolve(a,b)
print

fast_result = fft_convolve(a,b)
print 'fft convolve', fast_result
print 'rounded to real integers:', numpy.array([
    int(numpy.round(numpy.real(x))) for x in fast_result ])
```

13.7.1 Padding to power of two lengths

The Cooley-Tukey FFT only works when n is a power of 2; however, we can multiply two polynomials of arbitrary length by rounding the maximum of their lengths up to the next power of 2. This follows identically to the scheme for zero padding described above: a polynomial $a(x) = a_0 + a_1x + a_2x^2$ would be padded to $a(x) = a_0 + a_1x + a_2x^2 + 0x^3$, which has a coefficient vector of length four: $[a_0, a_1, a_2, 0]$. In a manner reminscent of the description of Strassen's algorithm in Chapter 12, this will, in the worst-case scenario, result in a runtime that will be bounded above by $r(2n)$, which will still be $\in \Theta(n \log(n))$.

13.8 Runtime of FFT circuits

We've already shown the runtime of FFT to be $\in \Theta(n \log(n))$. It is an extremely important open problem whether a standard, serial computer admits an FFT algorithm $\in o(n \log(n))$; however, that does not mean there are not other ways to speed up FFT in practice.

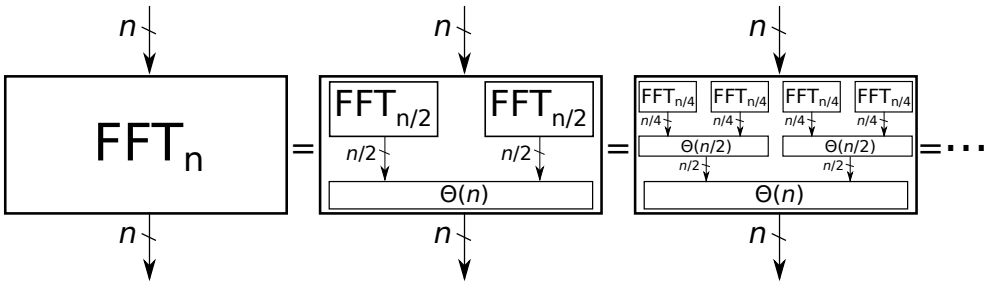

Figure 13.1: An FFT circuit. The circuit has n input wires and n output wires. The logical extreme of this circuit (if we continued unrolling every FFT at the $\cdots$ on the right side) will simply have $\log_2(n)$ of the postprocessing steps stacked up. Each layer of these postprocessing steps will perform total work $\in \Theta(n)$, but since they are performing element-wise operations, they could be solved simultaneously by separate, dedicated sub-circuits. The runtime of such a circuit will be limited by the propagation delay of the electricity through the layers, of which there are $\log_2(n)$; therefore, the runtime of this circuit would be $\in \Theta(\log(n))$.

One way to do so is to implement a constant-length FFT in circuitry. This is done just as it is in the recursive implementation. But in circuitry, both recursive calls can run simultaneously. This can be continued all the way down to the base case (Figure 13.1), achieving a circuit whose propagation delay is $\in \Theta(\log(n))$.

13.9 The "number theoretic transform"

At the point where we decided to use complex numbers to find points $x_i^2 = -x_{\frac{n}{4}+i}^2$, there actually are other options[3] At first glance, it may seem that complex numbers are the only possible solution, but we can actually do this with integers modulo some prime, also known as a "finite field". For example, if we are working modulo prime $p = 13$, and we choose $x_0 = 5$, then the constraint $x_0^2 \equiv -x_{\frac{n}{4}}^2 \pmod{13}$, has solution $x_{\frac{n}{4}} = 1$ (note that to negate a value in a finite field, we simply move backwards; *e.g.*, $-1 \pmod{13} =$

[3]We proved that complex numbers would be sufficient, but did not consider whether they were necessary.

$13 - 1 = 12$ and $12 + 1 \pmod{13} = 0 \pmod{13}$). Thus, we have $x_{\frac{n}{4}}^2 \equiv -x_0^2 \pmod{13}$.

An FFT implementation working in a finite field is called the "number theoretic transform" (NTT). Because it doesn't use floating point math, it can be used to perform exact convolutions of integer sequences, and is the basis of the Schönhage-Strassen algorithm for multiplying arbitrary-precision integers; that algorithm is a faster alternative to Karatsuba's method that we derived in Chapter 11. Because of it's greater speed, arbitrary-precision integers implemented using the NTT is a crucial ingredient in a solid cryptography library; that greater speed is even more important when attempting to crack a cryptographic key through brute force. Where FFT is well studied, the NTT is considerably newer and so there is currently still less literature and example code available.

Chapter 14

Subset-sum and Convolution Tree

Consider a restaurant that offers your family a free meal. The only condition is that you *must* spend exactly $1073.00. If you spend any other amount, you'll need to pay, but if your bill is exactly $1073.00, it will be free. We face two questions: the first question is whether, given the menu of the restaurant, it is even possible to spend exactly $1073.00; the second question is *which* items we should order so that we'll spend exactly $1073.00.

This problem is "the subset-sum" problem, and it is a quite famous (and surprisingly difficult) problem in computer science. Formally, subset-sum gives a set (here, the prices on the menu), M and a total amount t, and asks whether

$$\exists S \subseteq M : \sum_{i \in S} i = t;$$

that is, it asks whether there exists a subset of M named S such that the sum of all elements in S is t (note that there is a small subtlety that states that it is not valid to spend $1073.00 by ordering the same menu item multiple times).

14.1 Brute-force

The brute-force solution to subset sum is fairly obvious: we simply try all subsets of M and check whether one of them sums to t. We call the set of all subsets of M the "power-set" of M. If $M = \{m_0, m_1, \ldots m_{n-1}\}$, then we

can find the power set by enumerating every subset of M by how many items each subset contains:

$$
\begin{aligned}
powerset(M) = \{ & \\
& \{\}, \\
& \{m_0\}, \{m_1\}, \{m_2\}, \ldots \{m_{n-1}\}, \\
& \{m_0, m_1\}, \{m_0, m_2\}, \ldots \{m_1, m_2\}, \{m_1, m_3\}, \ldots \{m_{n-2}, m_{n-1}\}, \\
& \vdots \\
& M \setminus m_0, M \setminus m_1, M \setminus m_2, \ldots M \setminus m_{n-1}, \\
& M \\
\}. &
\end{aligned}
$$

In this manner, we need only iterate through every $S \in powerset(M)$, as shown in Listing 14.1.

Listing 14.1: Solving subset-sum with brute force. The output is `False True`, indicating that 110 cannot be made by any subset of M, but that 120 can be made by at least one subset of M.

```
import itertools

# from stackoverflow:
def powerset(items):
  s = list(items)
  return itertools.chain.from_iterable(itertools.combinations(s, r) for r
      in range(len(s)+1))

def powerset_subset_sum(M, t):
  for S in powerset(M):
    if sum(S) == t:
      return True
  return False

M = set([3, 5, 7, 9, 11, 13, 109, 207, 113, 300])

print powerset_subset_sum(M, 110)
print powerset_subset_sum(M, 120)
```

The runtime of a brute-force approach to subset-sum is at least exponential in $n = |M|$. We can see this in two ways. The first way is simple: each item in M can either be present or absent. Therefore, each contributes 1

bit of information. Together, the n items contribute n bits of information, and we are therefore enumerating through all possible bitstrings of length n. There are 2^n such bit-strings, and so the number of sets that we check must be $\in \Omega(2^n)$. The runtime must also be $\in \Omega(2^n)$, because the cost of checking the sum of any set S is nonzero, and so the runtime of the power-set method is bounded above by the number of sets that need to be checked.

14.2 Dynamic programming

We can improve this by using dynamic programming (mentioned in Chapter 7): here we will start our total at 0. Then, we will iterate through every value M and add it to the total. But rather than add it deterministically, we will add both outcomes: we will add the outcome where the element m_i is included in S and we will add the outcome where the element m_i is excluded from S. This is shown graphically in Figure 14.1.

As we move left to right, each layer of the graph will perform the Cartesian product between reachable nodes at the layer $i-1$ and the set $\{0, m_i\}$. The current totals that can be reached in each layer can be stored in a sparse manner (via set or dictionary[1]) or in a dense manner via array (as shown in Figure 14.1).

When asking whether

$$\exists S \subseteq M : sum_{i \in S} = t,$$

we need only determine whether there exists a path ending at the node with value t (*i.e.*, the node with a height of t) in the far right layer.

To figure out the cost of dynamic programming, we will restrict that each of the $m_i \leq k$ for some value k representing the menu item with maximum price. Now we will compute the computational cost in terms of both $n = |M|$ and k. Note that these are related to our goal t: $t \leq n \cdot k$, because that would be the maximum meal attainable when ordering one of every menu item (where each has price $\leq k$); as a result, sometimes you will see the computational complexity of dynamic programming for knapsack depicted in terms of n and t instead of n and k.

At each left-to-right layer in the graph, the height of the next layer will increase by at most k. The number of nodes in layer i, h_i will therefore be

[1] Also called a "map"

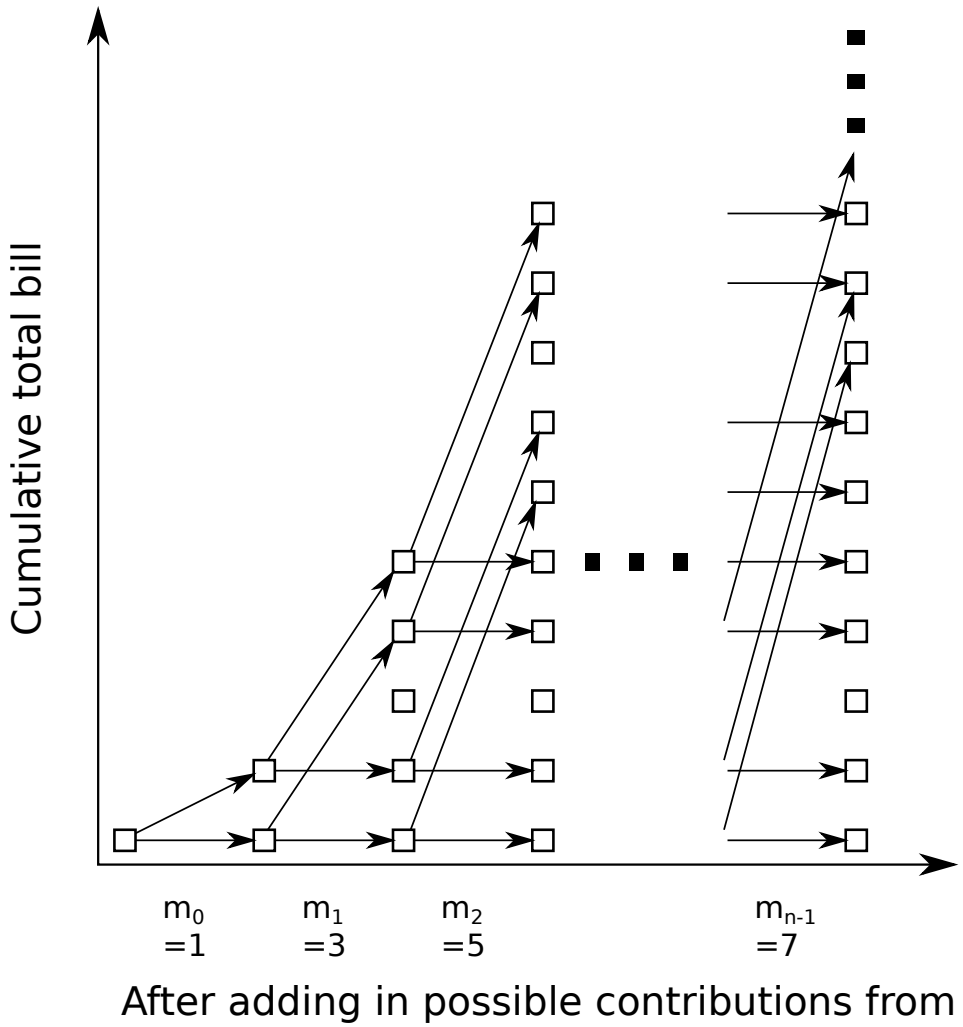

Figure 14.1: Dynamic programming for solving subset-sum on a set of integers. The total bill starts at 0 and each element of the set $M = \{m_0, m_1, m_2, \ldots\ m_{n-1}\} = \{1, 3, 5, \ldots\ 7\}$ is added in left-to-right. Each of these elements may be excluded from the set S (moving horizontally without moving up) or included in the set S (moving diagonally up and right, where we move up m_i units). For example, after adding in $m_0 = 1$ the total is in $\{0, 1\}$. After adding in m_1 the total is in $\{0, 1, 3, 4\}$.

$h_i \leq k \cdot i + 1$. Note that we add 1 because layer 0 has 1 node. Thus, creating these nodes will cost

$$\begin{aligned}
\sum_{i=0}^{n} h_i &\leq \sum_{i=0}^{n} (k \cdot i + 1) \\
&= \sum_{i=0}^{n} k \cdot i + \sum_{i=0}^{n} 1 \\
&= k \cdot \sum_{i=0}^{n} (i) + n \\
&\in k \cdot \Theta(n^2) + n \\
&= \Theta(k \cdot n^2).
\end{aligned}$$

This does not yet show that we have found a solution in $\Theta(k \cdot n^2)$. It only shows that we have built the nodes in that amount of time; however, the total runtime for solving subset-sum on a set of integers using this dynamic programming technique will not be substantially larger than the cost of initializing these nodes. The reason for this is that we add at most 2 edges out from every node (if the node was reachable, then we add an edge out excluding m_i from S and an edge out including m_i in S). Thus, the cost of adding edges and marking whether a node in the next layer is reachable (via a simple boolean) is no more than a constant factor greater than the total number of nodes. Thus, the total cost of our dynamic programming approach to subset-sum is $\in \Theta(k \cdot n^2)$. This may also be written as

$$\Theta(k \cdot n^2) = \Theta\left(\frac{t}{n} \cdot n^2\right) = \Theta(t \cdot n).$$

This approach is shown in Listing 14.2.

Listing 14.2: Solving subset-sum with dynamic programming. This method uses the same strategy as Figure 14.1. The output is `False True`, indicating that 110 cannot be made by any subset of M, but that 120 can be made by at least one subset of M.

```
def dynamic_programming_subset_sum(M, t):
  # start current_layer[0] as True
  current_layer = [True]

  for m_i in M:
```

```
    next_layer_size = len(current_layer) + m_i
    next_layer = [False]*next_layer_size

    for j in xrange(len(current_layer)):
      if current_layer[j]:
        next_layer[j + 0] = True
        next_layer[j + m_i] = True

    current_layer = next_layer

  # look in index t of current_layer and see whether or not it was
  # reachable
  return current_layer[t]

M = set([3, 5, 7, 9, 11, 13, 109, 207, 113, 300])

print dynamic_programming_subset_sum(M, 110)
print dynamic_programming_subset_sum(M, 120)
```

Note that this does *not* show that this is an algorithm that solves subset sum with runtime that is polynomial in the input parameters. The reason for this is that the value t requires only $b = \lceil \log_2(t) \rceil$ bits[2]. When considering the number of bits given, the runtime is $\in \Theta(2^b \cdot n)$. For this reason, we must be very careful when specifying whether or not an algorithm has a polynomial runtime[4]. Our dynamic programming approach to subset-sum has a runtime polynomial in n and k, polynomial in n and t, and at least exponential in b.

14.3 Generalized subset-sum

We will denote the case where each individual has their own custom menu, from which they can order any items as the "generalized subset-sum" problem. In this case, we will continue to denote the maximum amount each individual can spend as $k-1$ (*i.e.*, there are k distinct values $\{0, 1, 2, \ldots k-1\}$ that each individual may be able to spend). Likewise the goal value t also

[2]We would also need bits to specify the values in the set M, but here this is ignored for simplicity; however, each element is $\leq k$, and so we need roughly $n \cdot \lceil \log_2(k) \rceil$ bits additional for the set[3].

[3]We also may need some sort of delimeter to determine where one element's bits end and where the next element's bits begin.

[4]To do so, we need to specify polynomial *in what*?

does not need to have a fixed value; instead, it can be one of many allowed values specified by a set T. Here we are considering whether the sums of each individual's total purchases can reach any allowed goal $t \in T$:

$$m_0 + m_1 + m_2 + \cdots + m_{n-1} = t,$$

where $m_0 \in M_0, m_1 \in M_1, \ldots m_{n-1} \in M_{n-1}$, where $t \in T$, and where all $M_i \subseteq \{0, 1, 2, \ldots k-1\}$. For example, consider a possible generalized subset-sum problem with $n = 2$ and $k = 6$: individual 0 is able to spend any amount $\in \{0, 1, 5\}$ and individual 1 is able to spend $\in \{0, 1, 2, 4\}$. The total amount that can be spent by these two individuals is given by the Cartesian product between those sets:

$$\begin{aligned} &= \{i + j \mid \forall i \in M_0, \forall j \in M_1\} \\ &= \{0, 1, 2, 3, 4, 5, 6, 7, 9\}. \end{aligned}$$

This generalized subset-sum problem can be solved using the same dynamic programming technique as the standard subset-sum problem; however, now the runtime is no longer dominated by the number of nodes in each layer, but by the number of edges in the graph.

The edges between layer i and layer $i+1$ will be the product of the number of nodes in layer i and the changes that each can introduce. The number of nodes in layer i will be roughly the same as before $h_i \leq k \cdot i + 1$; however, where before we had at most two edges coming from each node, we now have at most k edges coming from each node. Thus, the total number of edges

that we need to insert and process will be

$$\begin{aligned}
\sum_{i=0}^{n-1} h_i \times k &= k \cdot \sum_{i=0}^{n-1} h_i \\
&\leq k \cdot \sum_{i=0}^{n-1} (k \cdot i + 1) \\
&= k \cdot \left(\sum_{i=0}^{n-1} k \cdot i + \sum_{i=0}^{n-1} 1 \right) \\
&= k \cdot \left(\sum_{i=0}^{n-1} (k \cdot i) + n \right) \\
&= k \cdot \left(k \cdot \sum_{i=0}^{n-1} (i) + n \right) \\
&\in \Theta(k^2 \cdot n^2).
\end{aligned}$$

Thus, we can use dynamic programming to solve the generalized subset-sum problem $\in \Theta(k^2 \cdot n^2)$.

14.4 Convolution tree

It is an interesting question whether or not we can solve the generalized subset-sum problem faster than $\Theta(k^2 \cdot n^2)$ used by the dynamic programming algorithm above. One thing that we can observe is that the dynamic programming starts efficient, but becomes slow as we progress and build the graph in a left-to-right manner. This is because the number of edges between a layer and the next layer will be bounded by the Cartesian product between the number of current nodes and the set $\{0, 1, 2, \ldots k-1\}$. The number of nodes in the leftmost layers (processed first) will be small, but as we progress, the Cartesian product becomes large.

One approach to attacking this problem is to merge arrays of similar size, thereby keeping them small as long as possible. For instance, the dynamic programming approach to summing the restaurant bill corresponds to placing the parentheses this way $(((M_0 + M_1) + M_2) + M_3$; instead, we can place the parenthesis in a manner that will merge pairs and then merge those merged

pairs: $(M_0 + M_1) + (M_2 + M_3)$. In this manner, we will merge collections of roughly equal size.

At first glance, this approach may appear promising; however, consider the final merger that will be performed: The final merger will be between $\frac{n}{2}$ people's subtotals and $\frac{n}{2}$ people's subtotals. Each will have subtotals $\in \Theta(k \cdot \frac{n}{2})$. Performing the Cartesian product will have a cost

$$\in \Theta\left(k \cdot \frac{n}{2} \times k \cdot \frac{n}{2}\right) = \Theta(k^2 \cdot n^2);$$

therefore, we know that the cost of processing the full graph will be $\in \Omega(k^2 \cdot n^2)$, which is no improvement.[5]

The limiting factor is thus a single Cartesian product merger. Recall that when merging sets A and B, we perform

$$C = A \oplus B = \{i + j \mid \forall i \in A, \forall j \in B\}.$$

This could be rewritten as a logical `AND`:

$$i \in A \ \wedge \ j \in B \rightarrow i + j \in C.$$

In turn, we could also rewrite this as a union of all possible ways to get to any sum $i + j$:

$$C = \bigcup_{i \in A, j \in B} \{i + j\}.$$

Let us now shift from thinking of sets to thinking of equivalent vector forms:

$$\begin{aligned} a_i &= \begin{cases} 1 & i \in A \\ 0 & i \notin A \end{cases} \\ b_i &= \begin{cases} 1 & i \in B \\ 0 & i \notin B \end{cases} \\ c_i &= \begin{cases} 1 & i \in C \\ 0 & i \notin C \end{cases}. \end{aligned}$$

[5] It *will* actually yield a small constant speedup, but we are looking for more than a constant improvement.

From this, we have

$$\begin{aligned} c_m &= \bigcup_{i,j:i+j=m} a_i \wedge b_j \\ &= \bigcup_i a_i \wedge b_{m-i}. \end{aligned}$$

Now, we can think of the union, $\bigcup_i$, as a weaker form of a $\sum_i$. That is, if we perform $\sum_i$ on some vector of booleans, we can compute the union by testing whether or not the sum was > 0; therefore, we have

$$\begin{aligned} d_m &= \sum_i a_i \wedge b_{m-i}, \\ c_m &= d_m > 0. \end{aligned}$$

Also note that with binary values a_i and b_j, the logical `AND` is equivalent to multiplication: $a_i \wedge b_j = a_i \cdot b_j$. Thus, we have

$$\begin{aligned} d_m &= \sum_i a_i \cdot b_{m-i} \\ &= a \circledast b, \\ c_m &= d_m > 0, \end{aligned}$$

where $a \circledast b$ denotes the convolution between vectors a and b.

In Chapter 13 we showed that a convolution between two vectors of length ℓ can be computed in $\Theta(\ell \log(\ell))$ by using FFT. Thus, we can use FFT convolution to merge these sets and perform $C = A \oplus B$ in $\Theta(\ell \log(\ell))$, where $\ell = \max(|A|, |B|)$.

If we put this together with the merging pairs strategy above, we construct a tree data structure known as the "convolution tree": In the first layer, there are n individuals (each described by a vector of length k) and $\frac{n}{2}$ pairs. We merge each of these pairs using FFT in runtime $\Theta(\frac{n}{2} \cdot k \log(k)) = \Theta(n \cdot k \log(k))$. In the second layer, there are $\frac{n}{2}$ vectors (each with length $\leq 2k$), which we can arrange into $\frac{n}{4}$ pairs. The runtime of merging these pairs with FFT will be $\in \Theta(\frac{n}{4} \cdot 2k \log(2k)) = \Theta(\frac{n}{2} \cdot k \log(2k))$. We can continue in this

manner to compute the total runtime:

$$\begin{aligned}
\sum_{i=0}^{\log_2(n)} \frac{n}{2^{i+1}} \cdot 2^i \cdot k \log_2(2^i \cdot k) &= \sum_{i=0}^{\log_2(n)} \frac{n}{2} \cdot k \log_2(2^i \cdot k) \\
&= \frac{n}{2} \cdot \sum_{i=0}^{\log_2(n)} k \cdot \big(\log_2(2^i) + \log_2(k)\big) \\
&= k \cdot \frac{n}{2} \cdot \sum_{i=0}^{\log_2(n)} (i + \log_2(k)) \\
&= k \cdot \frac{n}{2} \cdot \sum_{i=0}^{\log_2(n)} (i) + \sum_{i=0}^{\log_2(n)} (\log_2(k)) \\
&= k \cdot \frac{n}{2} \cdot \left(\frac{\log_2(n)(\log_2(n)+1)}{2} + \log_2(n)\log_2(k) \right) \\
&= k \cdot \frac{n}{2} \cdot \log_2(n) \left(\frac{(\log_2(n)+1)}{2} + \log_2(k) \right) \\
&\in \Theta(k \cdot n \log(k \cdot n) \log(n)).
\end{aligned}$$

This runtime is "quasilinear" meaning it is only logarithmically slower than the linear $k \cdot n$ cost of loading the data that defines the problem.

Thus far, we have used convolution for addition between variables that can each take several values. Above, we did this in terms of sets. We can also do this in terms of probability distributions, which are identical to the vector forms of sets, except for the fact htat probability distributions will be normalized so that the sum of the vector is 1. Such a normalized vector is called a "probability mass function" (PMF). An implementation of the convolution tree is shown in Listing 14.3. The successive merging of pairs is known as the "forward pass" of the convolution tree, and can be used to solve the generalized subset-sum problem in subquadratic time in both n and k. The probabilistic version of whether the party could possibly reach any total bill $t \in T$ and is indicated by whether the "prior" vector, which is produced by the final merger, is nonzero at the index corresponding to any goal value. This corresponds to the forward pass denoted in Figure 14.2.

Listing 14.3: Solving generalized subset-sum with the convolution tree. The priors (accumulated by the forward pass) and the likelihoods (accumulated

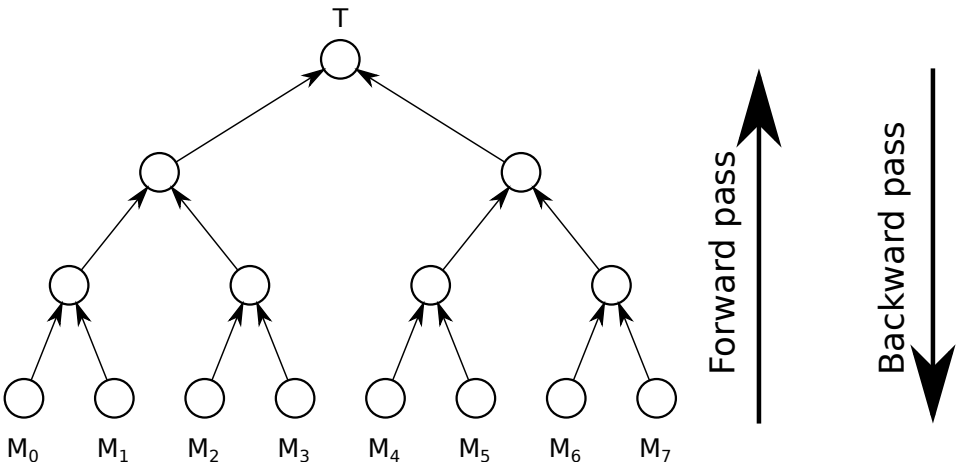

Figure 14.2: Convolution tree, solving generalized subset-sum in sub-quadratic time. A convolution tree for $n = 8$ is depicted. The forward pass finds whether any element of T can be built as a sum of elements from $M_0, M_1, M_2, \ldots$. The backward pass determines which elements of M_0, of M_1, *etc.*, were used to achieve valid sums $t \in T$.

by the backward pass) that are output are equivalent to the brute-force formulation.

```
import numpy
from scipy.signal import fftconvolve

class PMF:
  def __init__(self, start_value, masses):
    self._start_value = start_value
    self._masses = numpy.array(masses, float)
    self._masses /= sum(self._masses)

  def size(self):
    return len(self._masses)

  def narrowed_to_intersecting_support(self, rhs):
    start_value = max(self._start_value, rhs._start_value)
    end_value = min(self._start_value+self.size() - 1,
        rhs._start_value+rhs.size() - 1)
    masses = self._masses[start_value - self._start_value:]

    end_size = end_value - start_value + 1
    masses = masses[:end_size]
```

```
    return PMF(start_value, masses)

  def __add__(self, rhs):
    return PMF(self._start_value + rhs._start_value,
        fftconvolve(self._masses, rhs._masses))

  def __sub__(self, rhs):
    return PMF(self._start_value - (rhs._start_value+rhs.size()-1),
        fftconvolve(self._masses, rhs._masses[::-1]))

  def __mul__(self, rhs):
    this = self.narrowed_to_intersecting_support(rhs)
    rhs = rhs.narrowed_to_intersecting_support(self)

    # now the supports are aligned:
    return PMF(this._start_value, this._masses*rhs._masses)

  def support(self):
    return list(xrange(self._start_value, self._start_value+self.size()))

  def support_contains(self, outcome):
    return outcome >= self._start_value and outcome < self._start_value +
        self.size()

  def get_probability(self, outcome):
    return self._masses[outcome - self._start_value]

  def __str__(self):
    result = 'PMF('
    for i in xrange(self.size()):
      mass = self._masses[i]
      result += str(self._start_value + i) + ':' + str(numpy.round(mass,
          4))
      if i != self.size()-1:
        result += '\t'
    result += ')'
    return result

  def __repr__(self):
    return str(self)

import itertools
# runtime is \in \Omega(k^n)
def brute_force_solve(prior_pmfs, likelihood_pmf):
```

```
# prior_pmfs = [X_0, X_1, ...]
# likelihood_pmf = Y|D

# compute prior of Y:
prior_supports = [ pmf.support() for pmf in prior_pmfs ]
all_joint_events = itertools.product(*prior_supports)

prior_of_y = [0.0]*likelihood_pmf.size()
for joint_event in all_joint_events:
  y = sum(joint_event)
  if likelihood_pmf.support_contains(y):
    probability = numpy.product([ pmf.get_probability(event) for
        event,pmf in zip(joint_event,prior_pmfs) ])
    prior_of_y[y-likelihood_pmf._start_value] += probability

prior_of_y = PMF(likelihood_pmf._start_value, prior_of_y)

# compute likelihoods X_1|D, X_2|D, ...
likelihoods = []
for i in xrange(len(prior_pmfs)):
  priors_without_i = [ prior_pmfs[j] for j in xrange(len(prior_pmfs))
      if j != i ]
  distributions = priors_without_i + [likelihood_pmf]

  supports = [ pmf.support() for pmf in priors_without_i ] + [
      likelihood_pmf.support() ]
  all_joint_events = itertools.product(*supports)

  result_i = [0.0]*prior_pmfs[i].size()
  for joint_event in all_joint_events:
    y = joint_event[-1]
    sum_x_without_i = sum(joint_event[:-1])

    probability = numpy.product([ pmf.get_probability(event) for
        event,pmf in zip(joint_event,distributions) ])

    x_i = y - sum_x_without_i
    if prior_pmfs[i].support_contains(x_i):
      result_i[x_i - prior_pmfs[i]._start_value] += probability
  result_i = PMF(prior_pmfs[i]._start_value, result_i)
  result_i = result_i.narrowed_to_intersecting_support(prior_pmfs[i])
  likelihoods.append(result_i)

return likelihoods, prior_of_y
```

```
# runtime is \in \Theta(n k \log(n k) \log(n))
def convolution_tree_solve(prior_pmfs, likelihood_pmf):
  # prior_pmfs = [X_0, X_1, ...]
  # likelihood_pmf = Y|D
  n = len(prior_pmfs)

  # forward pass:
  layer_to_priors = []
  layer_to_priors.append(prior_pmfs)

  while len(layer_to_priors[-1]) > 1:
    layer = []
    for i in xrange(len(layer_to_priors[-1])/2):
      layer.append( layer_to_priors[-1][2*i] + layer_to_priors[-1][2*i+1]
          )
    if len(layer_to_priors[-1]) % 2 != 0:
      layer.append(layer_to_priors[-1][-1])
    layer_to_priors.append(layer)

  layer_to_priors[-1][0] =
      layer_to_priors[-1][0].narrowed_to_intersecting_support(likelihood_pmf)

  # backward pass:
  layer_to_likelihoods = [ [likelihood_pmf] ]

  for i in xrange(1, len(layer_to_priors)):
    # j is in {1, ... len(layer_to_priors) - 1}
    j = len(layer_to_priors) - i

    layer = []
    for k in xrange(len(layer_to_priors[j])):
      parent_likelihood = layer_to_likelihoods[-1][k]

      if 2*k+1 < len(layer_to_priors[j-1]):
        # this PMF came from two parents during merge step:
        lhs_prior = layer_to_priors[j-1][2*k]
        rhs_prior = layer_to_priors[j-1][2*k+1]

        lhs_likelihood = parent_likelihood - rhs_prior
        rhs_likelihood = parent_likelihood - lhs_prior

        lhs_likelihood =
            lhs_likelihood.narrowed_to_intersecting_support(lhs_prior)
        rhs_likelihood =
            rhs_likelihood.narrowed_to_intersecting_support(rhs_prior)
```

```
        layer.append(lhs_likelihood)
        layer.append(rhs_likelihood)
      else:
        # this PMF came from one parent during merge step (because
        # previous layer was not divisible by 2):
        layer.append(layer_to_priors[j-1][2*k])

    # todo: adapt where not multiple of 2

    layer_to_likelihoods.append(layer)
  layer_to_likelihoods = layer_to_likelihoods[::-1]

  # returns likelihoods X_0|D, X_1|D, ... and prior for Y
  return (layer_to_likelihoods[0], layer_to_priors[-1][0])

A = PMF(3, [0.5,0,0.5])
print 'A', A
print ''

B = PMF(1, [1,0,0])
print 'B', B
print ''

C = PMF(0, [0,0.5,0.5])
print 'C', C
print ''

D = PMF(4, [0,0.333,0.0,0.0,0.333,0.333])
print 'D', D
print ''

print 'brute force:'
likelihoods, prior = brute_force_solve([A,B,C], D)
print 'prior D', prior
print 'likelihoods A,B,C', likelihoods
print

print 'convolution tree:'
likelihoods, prior = convolution_tree_solve([A,B,C], D)
print 'prior D', prior
print 'likelihoods A,B,C', likelihoods
```

Just as we performed probabilistic addition on the PMFs using FFT (again, that is equivalent to the vector convolution performed above with

FFT), we can also perform probabilistic subtraction between random variables using FFT convolution. In this manner, the convolution tree can also be used to figure out *which* orders from each individual at your table yielded any allowed goal total bill $t \in T$. That step is referred to as the "backward pass" and also has runtime $\in \Theta(k \cdot n \log(k \cdot n) \log(n))$. The probabilistic version of whether the party could possibly reach an allowed total bill $t \in T$ while a given individual also ordered a particular item is determined by the "likelihood" vector for that indivdiual: where the likelihood vector for that individual is nonzero at the index corresponding to the price of the item, then it was possible for them to order the item and to reach a valid goal price $t \in T$.

Importantly, the convolution tree also opens the door for us to *weight* the different menu items. A set $A = \{0, 2\}$ is equivalent to the vector $a = [True, 0, True]$, which in turn can be written as $\text{pmf}_A = [0.5, 0, 0.5]$; however, the PMF form also allows us to weight these menu choices on a continuum: $\text{pmf}_A = [0.9, 0, 0.1]$ would correspond to the same set $A = \{0, 2\}$, but would indicate $\Pr(A = 0) = 0.9$ and $\Pr(A = 2) = 0.1$. This means we can use this form to encode information about how good a given solution would be. This will be useful for the "knapsack problem" in the next chapter.

Chapter 15

Knapsack and Max-convolution

Where the subset-sum problem simply asks *whether* it is possible to reach total t using a subset of M, the knapsack problem asks for the *best* possible way to reach total t. We can think of this as allowing each of the menu items from Chapter 13 to also contribute some amount of "happiness" to your order, and seeks to maximize the food order $S \subseteq M$ with $sum_{i \in S} = t$ that would yield the maximum happiness for your table:

$$knapsack(M, t) = \max_{S \subseteq M : \sum_{i \in S} = t} \sum_{i \in S} happiness(i).$$

15.1 Brute-force

Our brute-force approach to knapsack is quite similar to the brute-force approach to subset-sum: we will iterate through every subset S (by computing the power-set of M), and then of those that have total cost t, we will take the one with maximum total happiness (Listing 15.1).

Listing 15.1: Solving knapsack with brute force. The output is 14, indicating that the maximum happiness order for the table achieving total cost $t = 120$ yields a total happiness of 14.

```
import numpy
import itertools

# from stackoverflow:
def powerset(items):
  s = list(items)
```

```
  return itertools.chain.from_iterable(itertools.combinations(s, r) for r
      in range(len(s)+1))

def powerset_knapsack(M, t):
  max_happiness = -1
  for S in powerset(M):
    total_cost = sum([ cost for cost, happiness in S ])
    total_happiness = sum([ happiness for cost, happiness in S ])
    if total_cost == t and total_happiness > max_happiness:
      max_happiness = total_happiness

  if max_happiness == -1:
    print 'Warning: no subsets had sum', t, '(i.e., subset-sum of M, t
        was False)'
  return max_happiness

M_and_happiness = [(3,1), (5,10), (7,6), (9,2), (11,3), (13,8), (109,11),
    (207,4), (113,7), (300,18)]

print powerset_knapsack(M_and_happiness, 120)
```

15.2 Dynamic programming

Our dynamic programming approach to knapsack looks very similar to our dynamic programming approach to subset-sum; the main difference is that the $n + 1$-partite graph for dynamic prgoramming with subset-sum used unweighted edges. Now, we will replace those with weighted edges. The weight of each edge will correspond to the happiness contributed by that edge. Edges that move horizontally (which exclude some m_i) will have weight $+0$, whereas edges that move diagonally upwards will have weight $+happiness(i)$.

Where our previous dynamic programming for subset-sum considered only whether we could reach index t in the rightmost layer of the graph, we now want to compute the highest-weighted path in the graph from the far left node to the node at index t in the rightmost layer of the graph. Fortunately, this is a fairly easy modification to make. Before, we assigned a boolean to the nodes in the next layer using whether the previous layer was

reachable. As a set, the layer was written as

$$\begin{aligned} layer_{\ell+1} &= layer_\ell \oplus \{0, m_\ell\} \\ &= \{i + j \mid \forall i \in layer_\ell, \forall j \in \{0, m_\ell\}\ \}. \end{aligned}$$

As we saw in Chapter 14, subset-sum dynamic programming can be written in terms of convolution between two arrays, which will count the number of ways to reach any node in the graph. When the number is greater than zero, then it is possible to reach a node:

$$\begin{aligned} d_m &= \sum_i a_i \cdot b_{m-i} \\ &= a \circledast b, \\ c_m &= d_m > 0, \end{aligned}$$

where we use the array forms of the sets $layer_\ell$ and $\{0, m_\ell\}$:

$$\begin{aligned} a_i &= \begin{cases} 1 & i \in layer_\ell \\ 0 & i \notin layer_\ell \end{cases} \\ b_i &= \begin{cases} 1 & i \in \{0, m_\ell\} \\ 0 & i \notin \{0, m_\ell\} \end{cases}. \end{aligned}$$

Now, we will need to keep track of the best path reaching any node:

$$\begin{aligned} d_m &= \max_i a_i + b_{m-i} \\ c_m &= d_m > 0. \end{aligned}$$

A dynamic programming implementation for knapsack is demonstrated in Listing 15.2. The runtime will match the dynamic programming approach to subset-sum from Chapter 14: $r(n) \in \Theta(k \cdot n^2)$.

Listing 15.2: Solving knapsack with dynamic programming. As in the brute-force approach from Listing 15.1, the output is 14, indicating that the maximum happiness order for the table achieving total cost $t = 120$ yields a total happiness of 14.

```
def dynamic_programming_knapsack(M_and_happiness, t):
  # start current_layer[0] with "happiness" 0
```

```
  current_layer = [0]

  for m_i, happiness_i in M_and_happiness:
    next_layer_size = len(current_layer) + m_i
    # a value of -1 indicates the outcome is not possible:
    next_layer = [-1]*next_layer_size

    for j in xrange(len(current_layer)):
      if current_layer[j] != -1:
        # do not include m_i (add +0 to happiness):
        next_layer[j + 0] = max(next_layer[j + 0], current_layer[j] + 0)
        # include m_i (add +happiness_i to happiness):
        next_layer[j + m_i] = max(next_layer[j + m_i], current_layer[j] +
            happiness_i)

    current_layer = next_layer

  # look in index t of current_layer and see whether or not it was
  # reachable
  return current_layer[t]

M_and_happiness = [(3,1), (5,10), (7,6), (9,2), (11,3), (13,8), (109,11),
    (207,4), (113,7), (300,18)]

print dynamic_programming_knapsack(M_and_happiness, 120)
```

15.3 Generalized knapsack

Generalized knapsack is the knapsack-like version of generalized subset-sum:

$$\max\{happiness(m_0)+happiness(m_1)+\cdots+happiness(m_{n-1})|m_0+m_1+\cdots+m_{n-1}=t\},$$

where $m_0 \in M_0, m_1 \in M_1, \ldots m_{n-1} \in M_{n-1}$, where $t \in T$, and where all $M_i \subseteq \{0, 1, 2, \ldots k-1\}$. Generalized knapsack can be solved in a manner similar to generalize subset-sum. Just like the generalized subset-sum approach, the runtime of a dynamic programming approach to knapsack will be $r(n) \in \Theta(k^2 \cdot n^2)$.

15.4 Max-convolution trees

Max-convolution trees can be used to solve the generalized knapsack problem; however, where merging a pair of nodes in a standard convolution tree is solved by standard convolution. Merging two vectors a and b to solve subset-sum was solved by convolution trees using the following:

$$\begin{aligned} d_m &= \sum_i a_i \cdot b_{m-i} \\ &= a \circledast b, \\ c_m &= d_m > 0, \end{aligned}$$

where $a \circledast b$ denotes the convolution between vectors a and b. Using FFT, standard convolution has cost $\in \Theta(\ell \log(\ell))$, where ℓ is the length of the longer vector between a and b.

However, to solve knapsack, we no longer want to use standard convolution to merge nodes; instead, we want to use a type of "max-convolution".

15.5 Max-convolution

Where standard convolution is defined using the operations $(+, \times)$,

$$\begin{aligned} d_m &= a \circledast b \\ &= \sum_i a_i \cdot b_{m-i}, \end{aligned}$$

$(\max, \times)$ convolution is defined as

$$d_m = \max_i a_i \cdot b_{m-i}.$$

Note that here we are using a product between elements $a_i \cdot b_{m-i}$, whereas the max-convolution-like formula that occurs in the dynamic programming approach uses a sum between elements $a_i + b_{m-i}$. If we can solve one of these efficiently, we can easily solve the other efficiently by simply transforming the vectors by using logarithms or exponentials, which will transform $+$ operations into $\times$ operations and vice-versa.

Unfortunately, the approach that we used with Karatsuba and FFT were only defined for rings, mathematical spaces that support inverse operations;

however, max-convolution uses a "semiring": the max operation does not have an inverse operation. For example, if we have $z = x + y$, we can use only z and x to solve for y: $y = z - x$. On the other hand, if we know that $z = \max(x, y)$ we cannot necessarily use z and x to solve for y. Thus, the availability of fast convolution algorithms on these semirings is a hot and important question. Until 2006, no max-convolution algorithm $\in o(n^2)$ was known[1]. The algorithm published in 2006 is not substantially faster than naive: while it has a runtime $\in o(n^2)$, its runtime is $\notin O(n^{2-\epsilon})$.

15.6 Fast numeric max-convolution

Fast numeric max-convolution provides a numeric approximation for solving max-convolution. This enables us to use max-convolution trees without ever performing slow Cartesian products, as discussed in Chapter 14.

The numeric approach to max-convolution exploits the fact that the result at any index m can be defined as a maximum over a vector $u^{(m)}$:

$$\begin{aligned} d_m &= \max_i a_i \cdot b_{m-i} \\ &= \max_i u_i^{(m)}, \\ u^{(m)} &= [a_i \cdot b_{m-i} \mid \forall i]. \end{aligned}$$

Note that we have not yet improved the performance over the naive $\Theta(n^2)$ approach; we have simply introduced a need to store a temporary vector, which may even slightly decrease the performance; however, we can start to think about mathematical approximations for estimating the maximum value in the $u^{(m)}$ vector.

The maximum value of $u^{(m)}$ can be found using the ∞-norm, also known as the Chebyshev norm:

$$\begin{aligned} \max_i u_i^{(m)} &= \lim_{p \to \infty} \|u^{(m)}\|_p \\ &= \lim_{p \to \infty} \left(\sum_i \left(u_i^{(m)} \right)^p \right)^{\frac{1}{p}}. \end{aligned}$$

[1] It was first discovered by Bremner *et al.* and published in 2006

This can be approximated using a numerically large value for p rather than a limit as $p \to \infty$. We will denote this numerically large value as p^*:

$$\max_i u_i^{(m)} \approx \left(\sum_i \left(u_i^{(m)} \right)^{p^*} \right)^{\frac{1}{p^*}}, \quad p^* \gg 1.$$

If we expand this approximation back into the formula defining max-convolution, we get

$$\begin{aligned} d_m &= \max_i a_i \cdot b_{m-i} \\ &\approx \left(\sum_i \left(u_i^{(m)} \right)^{p^*} \right)^{\frac{1}{p^*}} \\ &= \left(\sum_i \left(a_i \cdot b_{m-i} \right)^{p^*} \right)^{\frac{1}{p^*}} \\ &= \left(\sum_i a_i^{p^*} \cdot b_{m-i}^{p^*} \right)^{\frac{1}{p^*}} \\ &= \left(a^{p^*} \circledast b^{p^*} \right)_m^{\frac{1}{p^*}} \\ d &\approx \left(a^{p^*} \circledast b^{p^*} \right)^{\frac{1}{p^*}}. \end{aligned}$$

Thus we see that we can simply take every element of a to the power p^*, every element of b to the power p^*, convolve using FFT convolution, and then take every element in the result to power $\frac{1}{p^*}$.

One interpretation for why this approximation works is that $z = x^{p^*} + y^{p^*}$ can behave both as a ring (allowing FFT convolution to be used) and as a semiring: Given z and x, we can solve for y. But also, $z^{\frac{1}{p^*}} \approx max(x, y)$ when $p^* \gg 1$.

When implementing this numerically, using a p^* too large will result in numerically unstable behavior. This is because $(x + y) - x$ will numericaly evaluate to 0 when $x \gg y$. Using `double` numbers, this will break down at some point when $\frac{y}{x} < \epsilon \approx 10^{-15}$ or so. Likewise, using a p^* too small will result in a poor approximation of the Chebyshev norm. There are some sophisticated solutions, but one simple solution is to use a small exponential series of p^* values. We know that the results will be roughly stable when the

result $y = (x + y) - x$ is at least $\epsilon \cdot x$. If we normalize the problems first so that the maximum values in the vectors a and b are both 1, then we are guaranteed a numerically stable result when the result at index m is $> \epsilon$. Thus, we will perform a few convolutions using various p^* and then at each index in the result, choose the result from the largest p^* that was numerically stable. This numeric approach is demonstrated in Listing 15.3.

Listing 15.3: Numeric max-convolution. A numeric approach to estimating max-convolution. Where the exact max-convolution output by this example is $0.02, 0.06, 0.16, 0.2175, 0.58, 0.3625, 0.2$, the fasts numeric approach yields result $0.02, 0.06000011, 0.1600363, 0.21778642, 0.58, 0.36540009, 0.2$.

```
import numpy
from scipy.signal import fftconvolve

def naive_convolve(x, y):
  x_n = len(x)
  y_n = len(y)
  result_n = x_n+y_n-1

  result = numpy.zeros(result_n)

  for i in xrange(x_n):
    for j in xrange(y_n):
      result_index = i+j
      result[result_index] = result[result_index] + x[i]*y[j]
  return result

def naive_max_convolve(x, y):
  x_n = len(x)
  y_n = len(y)
  result_n = x_n+y_n-1

  result = numpy.zeros(result_n)

  for i in xrange(x_n):
    for j in xrange(y_n):
      result_index = i+j
      result[result_index] = max(result[result_index], x[i]*y[j])
  return result

def numeric_max_convolve(x, y, log_p_max, epsilon=1e-10):
  x_n = len(x)
  y_n = len(y)
```

```
  result_n = x_n+y_n-1

  result = numpy.zeros(result_n)

  all_p = 2.0**numpy.arange(log_p_max)
  for p in all_p:
    result_for_p = fftconvolve(x**p, y**p)
    stable = result_for_p > epsilon

    result[stable] = result_for_p[stable]**(1.0/p)
  return result

x=numpy.array([0.1, 0.3, 0.8, 0.5])
y=numpy.array([0.2, 0.15, 0.725, 0.4])

print 'Naive sum-convolution', naive_convolve(x,y)
print 'FFT sum-convolution', fftconvolve(x,y)
print
print

print 'Naive max-convolution', naive_max_convolve(x,y)
print 'FFT sum-convolution under p-norms:'
for p in 2**numpy.arange(7.0):
  # call absolute value on each elemnet using fabs so that very small
  # numeric errors like -1e-32 have real 1/p roots (otherwise results
  # are complex and numpy will complain):

  # nonetheless, for larger p, numeric instability creeps in:
  print ' p='+str(p), numpy.fabs(fftconvolve(x**p,y**p))**(1.0/p)
print
print

print 'Numeric max-convolution', numeric_max_convolve(x, y, 10)
```

Where the fastest exact approach to max-convolution is $\notin O(n^{2-\epsilon})$, this numeric approach reduces to a small number of FFT convolutions, each of which cost $\in \Theta(\ell \log(\ell))$. If we try an exponential sequence of p^* values up to $p^* \leq p^*_{\max}$ we are guaranteed that the p^* used at each index will be at least half the maximum stable p^*. The runtime of this approach will be $\in \Theta(\ell \log(\ell) \log(p^*_{\max}))$. Although this approach is not exact, it is by far the fastest approach in practice. For this reason, it is used in with max-convolution trees to estimate solutions to the generalized knapsack problem with runtime. When $p^*_{\max}$ is a constant (and thus $\log(p^*_{\max})$ is a constant

that can be removed from the runtime), the runtime for solving the generalized knapsack problem with max-convolution trees and fast numerical max-convolution will be $\in \Theta(k \cdot n \log(k \cdot n) \log(n))$. Thus we can see it is at worst only a constant factor worse than the cost of standard convolution tree from Chapter 14. This runtime is quasilinear, meaning it is only logarithmically slower than the linear $k \cdot n$ cost of loading the data that defines the problem.

Faster approaches, numeric or exact, for max-convolution are important for several combinatoric problems.

Chapter 16

Reductions, Complexity, and Computability

16.1 Reductions

Problem A reduces to problem B when problem A could be solved efficiently if a fast solution to B is available. This can be written as $A \leq B$. For example, median reduces to sorting: if we have a fast algorithm for sorting, then we sort the list and then choose the middle element as the median. Likewise, we saw that convolution reduces to FFT: if we have a fast FFT algorithm[1], then we can perform convolution fast.

There are multiple kinds of reductions, each of which is determined by the operations that we are permitted to use. A reduction of A to B means we are allowed to try to solve problem A by preprocessing the inputs, calling a subtroutine to solve problem B (possibly multiple times), and then post-processing the results from those results from the calls to the subroutine that solves B. If we only call B once and the preprocessing and postprocessing each cost $\in O(1)$, then we can show that the reduction is a high-quality, constant time reduction. But if we need to call B $\log(n)$ times and preprocessing and postprocessing each cost $\in O(n)$, we would have a lower-quality reduction.

Reductions can also be used to prove that an problem has no fast solution. If $A \leq B$, then any proof that A is difficult to solve would also imply that B

[1]Recall that, as shown in Chapter 13, we can perform inverse FFT using FFT and conjugation.

is difficult to solve; if B were easy to solve, it would imply A can be solved easily, which would contradict the premise that A was difficult. Proving that a problem is difficult to solve is known as "hardness".

16.1.1 Subset-sum and knapsack

In Chapter 15, we saw that knapsack can be solved using a dynamic programming approach that was highly similar to the way we solved subset-sum in Chapter 14. Also, we may observe that knapsack seems intuitively harder than subset-sum, because it also includes the "happiness" values, whereas subset-sum used only booleans. We will use reductions to confirm our suspicion that subset-sum is not harder than knapsack.

Assume we are given a subset-sum problem defined by some set $M = \{m_0, m_1, \ldots m_{n-1}\}$ and goal t. Subset-sum seeks to find whether or not there is a subset of M, $S \subseteq M$ where $\sum_{i \in S} i = t$. We will now create a knapsack problem that will solve this subset-sum problem. Furthermore, we will create this knapsack problem with little pre-processing, little postprocessing, and where the size of the knapsack problem is not much larger than the size of the subset-sum problem that we want to solve. First, we will introduce "happiness" values for each of the items in M: $\forall i, happiness(i) = 1$. Thus, if there is a subset of M that achieves sum t, the knapsack result will achieve a total happiness ≥ 0; however, there is still the case where there exists *no* subset of M that would satisfy the subset-sum problem. In this case, the results of knapsack[2] is undefined. Therefore, we will introduce a guaranteed solution for achieving sum t.

For this, we will add one additional element to the M set in our knapsack problem: $m_n = t$. This will only increase the problem size by 1. Likewise, we will specify $happiness(n) = -\infty$. Now we are guaranteed the knapsack problem will always be well defined. When our subset-sum problem is solveable, our knapsack problem can reach t in at least 2 ways: via some subset of $\{m_0, m_1, \ldots m_{n-1}\}$ and via m_n. If the subset-sum problem should yield a `True` result, then our knapsack problem will use $\{m_0, m_1, \ldots m_{n-1}\}$, and the resulting happiness reported by the knapsack will not equal $-\infty$. On the other hand, if we can only achieve goal t by using m_n, our knapsack will produce result $-\infty$. Thus, a subset-sum problem on a set of size n reduces to a knapsack problem of size $n + 1$.

[2] As we have described it in Chapter 15.

16.1.2 Matrix multiplication and matrix squaring

In Chapter 12, we saw that matrix multiplication can be performed in in sub-cubic time. This was an artful process, and one where it's easy to imagine that there may be other yet unforseen approaches. Likewise, if we are only interested in multiplying a matrix with itself, it is a special case of matrix multlication. Where matrix multiplication is concerned with $C = A \cdot B$, matrix squaring is concerned only with special-case problems of the form $C = D \cdot D$. Because it is a special case, we might wonder whether or not matrix squaring can be done substantially faster than matrix multiplication.

Matrix squaring reduces to matrix multiplication: Clearly, matrix squaring reduces to matrix multiplication: if we can solve $C = A \cdot B$ for any matrices A and B, then we can solve $C = D \cdot D$ by choosing $A = D$ and $B = D$. However, this would only show the obvious result that matrix multiplication is not substantially faster than matrix squaring. To see if matrix squaring can be faster, we would like to try the opposite reduction: can we use matrix squaring to solve matrix multiplication?

Matrix multiplication reduces to matrix squaring: At first, this reduction seems tricky to do: after all, matrix multiplication should take two arguments A and B, while matrix squaring takes only one argument D. For this reason, to perform this reduction, we'd need to embed both A and B inside a larger D. For instance, if we let

$$D = \begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix},$$

then

$$\begin{aligned} D \cdot D &= \begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix}^2 \\ &= \begin{bmatrix} 0 \cdot 0 + A \cdot B & 0 \cdot A + A \cdot 0 \\ B \cdot 0 + 0 \cdot B & B \cdot A + 0 \cdot 0 \end{bmatrix} \\ &= \begin{bmatrix} A \cdot B & 0 \\ 0 & B \cdot A \end{bmatrix}. \end{aligned}$$

Thus, if we look in the top left quadrant of $D \cdot D$, we can find the solution of $A \cdot B$. We have just shown that matrix squaring can be used to solve matrix multiplication.

Thus, if matrix squaring were ever substantially faster than matrix multiplication, then we could use that faster matrix squaring algorithm to perform faster matrix multiplication. This contradicts the notion that matrix squaring can ever be substantially faster than matrix multiplication.[3]

16.1.3 APSP and min-matrix multiplication

The "all-pairs shortest paths" (APSP) problem computes the shortest paths from all nodes i to all nodes j. This can be solved using the Floyd-Warshall algorithm (Listing 16.1), which runs in $\Theta(n^3)$.

Just as we saw with max-convolution in Chapter 15, we can swap some operations for others (in max-convolution, we replaced the $+$ operation with a max operation). Min-matrix multiplication is a similar modification of standard matrix multiplication. Specifically, $(\min, +)$ matrix multiplication performs min wherever standard matrix multiplication performs $+$ and performs $+$ everywhere standard matrix multiplication performs $\times$. A sample implementation of $(\min, +)$ matrix multiplication is shown in Listing 16.2.

Listing 16.1: Solving APSP with Floyd-Warshall. This algorithm runs in $\Theta(n^3)$ time.

```
import numpy

def floyd_warshall(A):
  n = A.shape[0]
  assert( A.shape == (n,n) )

  # make a copy of A:
  result = numpy.array(A)

  for k in xrange(n):
    for i in xrange(n):
      for j in xrange(n):
        result[i,j] = min(result[i,j], result[i,k]+result[k,j])

  return result

if __name__=='__main__':
  n=8
  numpy.random.seed(0)
```

[3] In practice, it may be $4\times$ faster or so, but it will not be asymptotically faster by more than a constant.

```
  A = numpy.random.randint(1, 16, n*n).reshape( (n,n) )
  # set diagonal distances to 0:
  for i in xrange(n):
    A[i,i] = 0

  print A
  print
  print floyd_warshall(A)
```

Listing 16.2: Min-matrix multiplication. This algorithm runs in $\Theta(n^3)$ time.

```
import numpy

def min_plus_matrix_product(A, B):
  n = A.shape[0]
  assert( A.shape == (n,n) and B.shape == (n,n) )

  result = numpy.zeros( (n,n) ) + numpy.inf
  for i in xrange(n):
    for j in xrange(n):
      for k in xrange(n):
        result[i,j] = min(result[i,j], A[i,k]+B[k,j])
  return result

if __name__=='__main__':
  n=4
  numpy.random.seed(0)
  A = numpy.random.randint(1, 16, n*n).reshape( (n,n) )
  # set diagonal distances to 0:
  for i in xrange(n):
    A[i,i] = 0

  print A
  print min_plus_matrix_product(A,A)
```

APSP reduces to $(\min, +)$ matrix multiplication: Given A, the $n \times n$ adjacency matrix describing a graph, let's consider the $(\min, +)$ matrix multiplication $A \cdot A$. The result $C = A \cdot A$ would be defined

$$C_{i,j} = \min_k A_{i,k} + A_{k,j}.$$

If we consider C as the adjacency matrix of a new graph, we can see that the edge $C_{i,j}$ is permitted to go from vertex i to vertex j by passing through any

vertex k: $C_{i,j}$ considers first edge $A_{i,k}$ and then second edge $A_{k,j}$. We call this an "edge contraction". Now consider that, given an adjacency matrix A with nonnegative distances, the shortest path from any vertex i to any vertex j will take at most n edges. This is because the longest non-cyclic path in a graph with n nodes will take at most n edges, and because we will never prefer to take a longer path (because the nonnegative edge weights can make the path no shorter). As a result, performing n edge contractions will solve APSP, because it will consider all paths between every starting node i to every ending node j.

We can perform this via n $(\min, +)$ matrix multiplications: the APSP adjacency matrix result will be given by A^n. Now, recall that in Chapter 7, we showed that we can memoize to compute a^b in roughly $\log(b)$ multiplications. The same can be done to compute A^n using roughly $\log_2(n)$ $(\min, +)$ matrix multiplications. Thus we see that APSP reduces to $\log(n)$ $(\min, +)$ matrix multiplications. We can see this in Listing 16.3.

Listing 16.3: Reducing APSP to $\log(n)$ min-matrix products.

```
from min_plus_matrix_product import *
from floyd_warshall import *

n=8
numpy.random.seed(0)
A = numpy.random.randint(1, 16, n*n).reshape( (n,n) )
# set diagonal distances to 0:
for i in xrange(n):
  A[i,i] = 0

print 'Adjacency matrix:'
print A
print

print 'APSP:'
result = floyd_warshall(A)
print result

print 'APSP with edge relaxations:'
new_result = A
for iteration in xrange(int(numpy.ceil(numpy.log2(n)))):
  new_result = min_plus_matrix_product(new_result, result)
print new_result
print 'maximum error', max( (result - new_result).flatten() )
```

$(\min, +)$ **matrix multiplication reduces to APSP:** Surpisingly, we can also reduce $(\min, +)$ matrix multiplication to APSP. Consider the $(\min, +)$ matrix product $C = A \cdot B$. We have

$$C_{i,j} = \min_k A_{i,k} + B_{k,j}.$$

We can construct a tripartite graph where the APSP solution will compute each $C_{i,j}$. This graph is constructed using two layers of edges: the first layer of edges uses edge weights taken from A and the second layer of edges uses edge weights taken from B (Figure 16.1).

Note that this is not as strong of a result as our circular reduction between matrix multiplication and matrix squaring. With matrix multiplication and matrix squaring, each problem reduced to one copy of another problem of a similar size. In contrast, in this case, APSP and min-matrix multiplication reduce to the other, but by using multiple copies of the other problem (seen when reducing APSP to min-matrix multiplication) or by significantly changing the size of the problem (seen when reducing min-matrix-product to APSP).

16.1.4 Circular reductions

Above, we saw that matrix squaring and matrix multiplication both reduce to one another. Because each of these reductions is efficient and does not increase or decrease the problem size by much (the problem size grows by a factor of 2 when we reduce matrix multiplication to matrix squaring), then we can say that these problems have roughly equivalent complexity.

Any collection of problems that are equivalent can help us describe set of problems with equivalent complexity. For example, if we ever prove that some new problem X reduces to matrix squaring, then we know it is in the worst-case scenario the same complexity as both matrix squaring and matrix multiplication; however, in an optimistic world, that problem X may still be easier than matrix multiplication. For instance, we saw above that median reduces to sorting; however, median can be solved in $\Theta(n)$ using a divide-and-conquer algorithm, and so median can be solved more easily than comparison-sorting, which we saw in Chapter 6 is $\in \Omega(n \log(n))$. If matrix multiplication also reduces to problem X, then we know that X is equivalent to matrix multiplication and matrix squaring.

Equivalent complexity can be a very useful tool for learning about a new problem; even if we see a new problem X on which little literature is

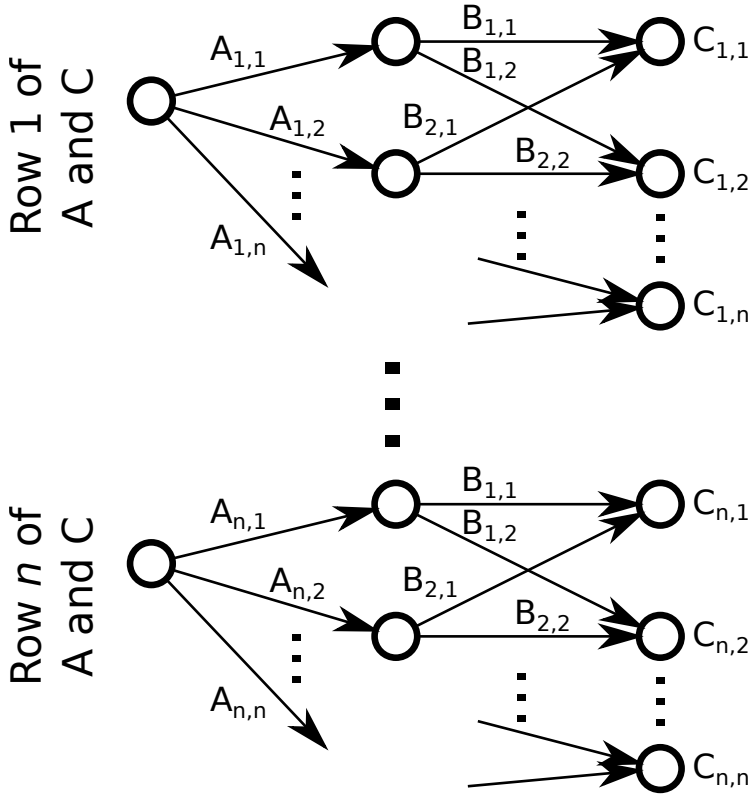

Figure 16.1: Min-matrix multiplication reduces to APSP. This graph uses three layers of nodes and two layers of edges. The first layer of edges uses weights from A and the second layer of edges uses weights from B. Importantly, any node in the middle layer reachable through an edge $A_{i,k}$ can only reach the right layer of nodes through edges of the form $A_{k,j}$. In this manner, the shortest path from the left layer of nodes to the node labeled $C_{1,2}$ will consider every path of the form $A_{i,k} + B_{k,j}$. The APSP solution therefore solves min-matrix product $C = A \cdot B$.

avialable on on which no fast algorithm is known, a circular reduction of $X \leq$ matrix multiplication and matrix multiplication $\leq X$ proves that X is neither hardar nor easier than matrix multiplication. Instantly, without designing an algorithm specifically for problem X, we may already achieve bounds on *all possible* algorithms for problem X. For instance, we would already know that any algorithm that solves X must be $\in \Omega(n^2)$ (because matrix multiplication needs to at least read in each of the n^2 input values) and that it should be possible to solve X in $O(n^{\log_2(7)})$.

16.2 Models of computation

16.2.1 Turing machines and nondeterministic Turing machines

Turing machines are a very useful model of computation, which include a finite state machine (FSM), which can read from and write to an infinite tape. Anything computable by the real RAM model of computation, which closely resembles how most modern computers work with fast random access to any cell in memory, can be computed by some Turing machine.

Unlike standard Turing machines or standard modern computers, nondeterministic Turing machines are able to take multiple branches simultaneously. Where a standard Turing machine or a standard modern computer follows one deterministic path in its FSM, a nondeterministic Turing machine can take two paths simultaneously. This ability is reminiscent of an ability to spawn new spawn threads without any limit or runtime penalty.

16.2.2 Complexity classes

A complexity class is a class of problems that can be solved in some runtime on a certain type of computer. The class P (for polynomial) consists of all problems that can be solved in polynomial time (*i.e.*, $\in O(n^k)$ for some constant k) using a standard Turing machine. An example of a problem in P is sorting, which we saw can be run in $O(n^2)$ in chapter 4.

The class NP (for nondeterministic polynomial) consists of all problems that can be solved in polynomial time using a nondeterministic Turing machine. Equivalently, the class NP is the class of problems that can be checked in polynomial time on a standard Turing machine. Sorting is a problem in

NP because we can check whether or not sorting has been performed by verifying whether or not $x_i \leq x_{i+1}$ in polynomial time using a standard Turing machine. Another problem in NP is boolean satisfiability. Boolean satisfiability checks whether a given logic circuit can be made to return `True` using some inputs.

16.2.3 NP-completeness and NP-hardness

The hardest problems in a complexity class are known as "complete". One very important class is NP-complete, which describes the hardest problems in class NP, *i.e.*, the hardest problems that can be checked in polynomial time. One of these problems that has been thoroughly studied is boolean satisfiability. Boolean satisfiability checks whether some logic circuit can be made to produce a `True` result. Any problem in NP will return a `True` or `False` by checking some result in polynomial time. Thus any problem in NP that can be run on a standard computer must do so by using logic gates in that computer, and so any problem in NP can be reduced to boolean satisfiability using some circuit from that standard computer[4]. This means that boolean satisfiability must be NP-complete. Therefore, a problem X to which boolean satisfiability reduces (*i.e.*, a problem X that could be used to solve boolean satisfiability) must also be able to solve any problem in NP; therefore such a problem X must also be NP-complete.

It is currently unknown whether or not P=NP or whether P≠NP. If P≠NP, then there must be some problem in NP that is not in P, *i.e.*, it must mean that there is some problem that is fast to verify, but which is not fast to solve. Equivalently, this means that there is no polynomial time algorithm that solves an NP-complete problem in polynomial time. This is an extremely important problem in computer science. One common way to show that a problem X has no known polynomial time solution is to reduce problem X to an NP-complete problem to it, meaning that a fast solution to X would imply a fast solution to some NP-complete problem, which would not be possible unless P=NP.

A problem that is the same difficulty or harder than a complete problem is known as "hard". For example, NP-hard refers to the problems that are at least NP-complete (*i.e.*, NP-complete or maybe harder).

[4]This is the substance of the Cook-Levin theorem.

16.3 Computability

Thus far in this book, we have been concerned primarily with fast ways to solve problems; however, even problems that are very slow can at least be solved with enough time. "Computability" refers to the distinction between problems for which there is a solution versus problems for which no solution can exist.

16.3.1 The halting problem

A classic example of an uncomputable problem is the "halting problem". Consider some program `DOES_IT_HALT(A, I)`, which takes two parameters, an algorithm A and an input I and decides whether calling algorithm A with input I will halt or whether it will run forever. Having a program like `DOES_IT_HALT` would be very useful for analyzing new algorithms or finding bugs.

Now let's consider whether such a program would even be possible to write. We will construct a new program, called `CONTRARION(X)`, which does the opposite of whatever `DOES_IT_HALT` tells it to do:

```
if DOES_IT_HALT(X, X) while (True) {} else halt.
```

What will happen when we call `CONTRARION(CONTRARION)`? If `CONTRARION(CONTRARION)` halts, then that means `DOES_IT_HALT(CONTRARION, CONTRARION)` must return `False`, which contradicts the fact that `CONTRARION(CONTRARION)` halts. On the other hand, if `CONTRARION(CONTRARION)` loops forever, then `DOES_IT_HALT(CONTRARION, CONTRARION)` must return `True`, which contradicts the notion that `CONTRARION(CONTRARION)` was just found to halt. Thus, finding a program that will determine if any other program will halt is not possible. We call such a problem "uncomputable"[5].

[5]This is reminiscent of Russel's paradox: If a barber is a man who shaves every man who doesn't shave himself, then does the barber shave himself? If he does, then he's shaving a man who shaves himself (contrary to the description of him). If he does not, then he *should* shave himself, because he is supposed to shave *every* man who doesn't shave himself.

16.3.2 Finite state machines and the halting problem

Modern computers are FSM (assuming they are not connected to an infinitely growing network or infinite external storage). All deterministic FSMs follow a property that guarantees that state S_i will always produce some state S_j. In this manner, if all of the n possible states for that computer are $S_1, S_2, \ldots S_n$, then any program that runs for more than n iterations must have reached some state S_k more than once. If this is true, then the computer will deterministically produce the loopy that brought it to state S_k again and again in an endless loop.

For this reason, it *would* be possible to solve the halting problem on any modern computer (since it is a deterministic FSM). We would simply create a table of all n possible states and then record whether or not those states eventually yield a state that's already been visited. This is essentially the idea of running every possible program and any program that runs for more than n iterations must be in an infinite loop.

This FSM excecption to the halting problem is because the computer is finite, and so constructing a `CONTRARION` program that contains `DOES_IT_HALT` (and so is larger than `DOES_IT_HALT`) is impossible. Note that even though it may be theoretically possible to solve the halting problem on a determinsitic FSM, it would not be practically feasible as the number of states n for a modern computer would be exponential in the number of bits that can be manipulated (including registers, RAM, and disk storage), and thus it would be extremely large.