Algorithms Notes

Simon Xiang

Lecture notes for the Spring 2022 section of Algorithms and Complexity (CS 331) at UT Austin, taught by Dr. Price. These notes were taken live in class (and so they may contain many errors). Source files: https://git.simonxiang.xyz/math_notes/files.html

Contents

1 Preliminary material

I am not a CS major. Here is some reading I had to do to catch up.

1.1 Basic algorithm analysis

Big-Oh notation (asymptotic notation) is used to analyze runtimes of algorithms. For a function $f(n)$, the notation $O(f(n))$ denotes the set of functions

$$O(f(n)) = \{g(n) \mid \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq c \cdot f(n) \text{ for all } n \geq n_0\}$$

In essence, big-O is the upper bound for f. We use asymptotic notation to simplify functions, for example in place of $f(n) := 5n\log n + 8n - 200$[1] we write $O(n\log n)$. This is proven as follows: $5n\log n + 8n - 200 \leq 5n\log n + 8n \leq 5n\log n + 8n\log n$ (for $n \geq 2$, ensuring $\log n \geq 1$) $\leq 13n\log n$. So $f(n) \in O(n\log n)$ with $c = 13, n_0 = 2$. Easy things: $O(n^{c_1}) \subset O(n^{c_2})$ for $c_1 < c_2$. For $a, b, c > 0$ constants we have

$$O(a) \subseteq O(\log n) \subseteq O(n^b) \subseteq O(c^n).$$

These hold when multiplied by anything positive, for example multiplying by n gives

$$O(n) \subseteq O(n\log n) \subseteq O(n^{1+b}) \subseteq O(nc^n).$$

We write things like $f_1(n) = O(f(n))$ or "$f_1(n)$ is $O(f(n))$" or "the runtime of $f_1(n)$ is $O(f(n))$" when we really mean $f_1(n) \in O(f(n))$. Sometimes we come across things like $T(n) = 2\log n + O(1)$, which means $T(n) \leq 2\log n + [\text{some member of } O(1)]$.

$\Omega(n)$ gives a lower bound, and $\Theta(n)$ means f is both in $O(n)$ and $\Omega(n)$ (tight bound).

todo:insertion sort, mergesort, quicksort, bubble sort

1.2 Substitution method

The process is simple:

(1) Guess the solution.

(2) Use induction to find the constants and show that the solution works.

"Substitution" comes from the fact that we substitue the guessed solution for the function when applying the inductive hypothesis to smaller values. This is very good but the caveat is we have to guess the function.

Example 1.1. To determine an upper time bound on $T(n) = 2T(\lfloor n/2 \rfloor) + nT(n) = 2T(\lfloor n/2 \rfloor) + n$, we guess that the solution is $T(n) = O(n\log n)$. We need to prove that $T(n) \leq cn\log n$ for some $c > 0$. Assume this holds for all todo:?? trick: substitute $m = \log n$

1.3 Recurrence trees

Recursion trees allow us to guess something reasonable to apply the substitution method to. For example, consider $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. Ignoring floors and ceilings, we create a tree for $T(n) = 3T(n/4) + cn^2$ for $c > 0$. Assume n is an exact power of four (sloppiness for convenience) so all subproblem sizes are integers. Refer to todo:figure. Part (a) shows $T(n)$, (b) shows expanding the tree into an equivalent recurrence. (c) shows expanding each node with cost $T(n/4)$ todo:finish

[1] In computer science, log is shorthand for $\log_2$ unless otherwise stated.

1.4 The master method

The master method (or theorem) helps solve recurrences of the form $T(n) = aT(n/b) + f(n)$, given $a \geq 1$ and $b > 1$ are constants, $f(n)$ is asymptotically positive.

Master Theorem. *Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$, where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the asymptotic bounds:*

(1) If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

(2) If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.

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

todo:something about polynomially larger

Example 1.2. Consider $T(n) = 9T(n/3) + n$. Here $a = 9, b = 3, f(n) = n$, and $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9 - \varepsilon})$ for $\varepsilon = 1$, we apply case one of the master theorem and conclude $T(n) = \Theta(n^2)$.

Example 1.3. Consider $T(n) = T(2n/3) + 1$, where $a = 1, b = 3/2, f(n) = 1$, $n^{\log_{3/2} 1} = 1$. Here case two applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$. So $T(n) = \Theta(\log n)$.

2 Algorithms and complexity

Complexity is the broad classification of different problem types (problems that can be solved by computers). We will spend the first three months of class in polynomial time P, specifically the time within P. For example, we have insertion sort (complexity $O(n^2)$) vs merge sort (complexity $O(n \log n)$).

Today we look at a way to reuse our computation to do things efficiently, called dynamic programming. Our first example is a classical algorithm we have been using our entire lives.

2.1 Multiplication

I'm not writing this out but we multiplied 331×388 using the algorithm we learned in elementary school. Then we added two large numbers which was much easier. So adding two n-digit numbers is $O(n)$ while multiplying results in n^2 digits, so adding them up means that multiplication is $O(n^2)$.

The reason why we don't care about constants in algorithms is that different operations take different amounts of time for different computers. A human would have a larger constant than a computer, the limiter may be writing down the numbers or low RAM. Who knows. The point is, relative time varies so we use big-O notation, which doesn't care.

We have a better algorithm for multiplication than schoolbook multiplication, called **Karatsuba multiplication**. Consider the numbers $A = 5124758048$ and $B = 617586337$. Split A and B into two halves as below:

$$\underbrace{51247}_{A_1}\underbrace{58048}_{A_2} \times \underbrace{61725}_{B_1}\underbrace{86337}_{B_2}$$

Then $A \cdot B = (10^{n/2}A_1 + A_2) \cdot (10^{n/2}B_1 + B_2) = 10^n \cdot A_1B_1 + 10^{n/2}(A_1B_2 + A_2B_1) + A_2B_2$. We have n-digit multiplication equal to four instances of $n/2$ digit multiplication plus three additions, so $T(n) = 4T(n/2) + O(n)$ (where T represents n-digit multiplication). We solve this using the **Master theorem**, which says $T(n) = aT\left(\frac{n}{b}\right) - f(n)$. Here $a = 4, b = 2, f(n) = O(n)$, so $c_{\text{crit}} = \log_b a = 2$. Case 1 implies that $T(n) = \Theta(n^{c_{\text{crit}}}) = \Theta(n^2)$.

The total work is the number of nodes times the work of each notes. As each level we multiply by an order of 4. Each node is working at an order of $\frac{n}{2^k}$, so the total work is $4^k \cdot \frac{n}{2^k} = 2^k n$. In short, $T(n) = 4T(n/2) + O(n) = O(n^2)$. Note that

$$(A_1 + A_2)(B_1 + B_2) = A_1B_1 + (A_1B_2 + A_2B_1) + A_2B_2 \implies A_1B_2 + A_2B_1 = (A_1 + A_2)(B_1 + B_2) - A_1B_1 - A_2B_2.$$

So instead of four $\frac{n}{2}$ terms we have three. Then $T(n) = 3T(\frac{n}{2}) + O(n)$, running it down means that the total work is now $\left(\frac{3}{2}\right)^k n$. So our time complexity is now $\left(\frac{3}{2}\right)^{\log_2 n} = n^{\log_2 \frac{3}{2}}$, which implies $T(n) = 3T\left(\frac{n}{2}\right) + O(n) = n^{\log_2 3} \approx n^{1.585}$ (also follows by Master theorem).

3 Recursion

Recursion is a very common idea used to write correct programs. It means we solve the problem by "recursively" solving smaller versions of the *same* problem. We have already seen this in various examples;

- Multiplication, as in the last class.
- Merge sort- we want to sort an array X. We sort the first half, then sort the second half, then merge. This is recursive because we have a problem, and we call the same function on a smaller instance. I
- Towers of Hanoi- see the book.

3.1 The n Queens problem

Consider the n queens problem. We have an n by n chessboard; can we place queens on the board such that they don't attack each other? The first question is, how many queens? Note that this is capped at n, since we only have n rows. When placing queens and it fails, we can go back and try other solutions, this leads to *backtracking*. Forgetting runtime, how would we implement this problem?

The strategy is as follows:

(1) Recast each answer as a *series of single choices*; (Row 1 Q, Row 2 Q, Row 3 Q, ...)

(2) Each successive choice is its own recursive call,

(3) If stuck, return.

Returning to the n queens problem.

```
    def collide(a, b):
    # a= (row, col), b= (row, col)

    return a[0] == b[0] or a[1] == b[1] or abs (a[0] - b[0]) == abs(a[1] - b[1])

def queens4():
    for q1 in range(4):
        for q2 in range(4):
            #Does placing a queen at (2, q2) collide with (1, q1)?
            if collide((1,q1), (2, q2)):
                continue
            for q3 in range(4):
                # Does placing a queen at (3, q3) collide with EITHER (1,q1) or (2,q2
                                                                    )?

                if collide((1, q1), (3, q3)) or collide((2, q2), (3, q3)):
                    continue
```

```
                for q4 in range(4):
                    if (collide((1, q1), (4, q4)) or collide((2, q2), (4, q4))
                        or collide((3, q3), (4, q4))):
                        continue
                    print(q1, q2, q3, q4)

queens4()
```

todo:fix code Okay, we've solved the four queens problem. But if we were to do this for eight queens, we would need eight nested loops. This is a huge pain. If we want to do this with general n, doing it iteratively is not a good solution at all.

```
    ok
```

todo:dicussion on just returning the first one, exponential time. This is the general strategy of recursion with backtracking, and is pretty generic.

3.2 Game Tree Evaluation

The question is "who wins at `[game]`"? Could be chess, checkers, tic-tac-toe, etc. We have a game state, say for white, and white has many different moves. For each board state, we have many different moves, and so on. todo:figure

Most of these alternate, but in principle but they don't have to. At the end we reach some position, say checkmate (for black). In this case black wins. How do we figure out who wins? (Suppose there are no draws). Say we are in the position `(player, state)`.

- If the state is now checkmate, we know who wins.
- If we are not in checkmate, we have some moves we can do, each leading to another state.
- Else, `winner(player, state)` wins if there exists any move to `state'` where `winner(other player, state')=player`.
- The other player wins otherwise.

We can apply this to draws with some sort of maximum or minimum score. We write this out as a recursion where for each player we try all possibilities and evaluate. If any of them lead to checkmate, that's a good move and we do it.

So writing a program to evaluate the winner is easy (computable), but finding a winner is hard.

3.3 Subset Sum

In this problem, we are given a set of positive integers X, say $[1,4,5]$, and a target T (say 9). The question is this: Find a subset of X that sums to T. This is a natural building block in a number of problems. How do we implement this? (Forget time and all of that).

todo:code

Listing out all subsets is hard.

```
    def subsetsum(X, T):
    """Find a subset of X that sums to T.

    Inputs:
        X: a list of positive integers, e.g. [1, 4, 5]
        T: a target positive integer, e.g. 9
```

```
    Outputs:
        None or a list that sums to T (e.g., [4, 5])
    """

    #base cases
    if T == 0:
        return []
    if not X or T < 0:
        return None

    #recursion
    take = subsetsum(X[1:], T - X[0])
    if take is not None:
        return [X[0]] + take
    skip = subsetsum(X[1:], T)
    return skip

print(subsetsum([1, 4, 5], 9))

import random
vals = [random.randint(10000, 20000) for _ in range(5)]
T = sum(v * random.randint(0, 1) for v in vals)
print(f"Instance: {vals} {T}")

print((subsetsum(vals, T)))
```

This code finds *a* solution. What if we want to find a *small* solution? todo:code This solution would be cleaner if we didn't have to rewrite our huge array each time. We do this by rewriting with an index.

```
    def subsetsum(X, T, i=0):
    """Find a subset of X that sums to T.

    Inputs:
        X: a list of positive integers, e.g. [1, 4, 5]
        T: a target positive integer, e.g. 9
        i: index we're looking at [so X[i:] is of interest]

    Outputs:
        None or a list that sums to T (e.g., [4, 5])
    """

    #base cases
    if T == 0:
        return []
    if len(X) <= i or T < 0:
        return None

    #recursion
    take = subsetsum(X[1:], T - X[0])
    if take is not None:
        return [X[0]] + take
    skip = subsetsum(X[1:], T)
    return skip

print(subsetsum([1, 4, 5], 9))
```

```
import random
vals = [random.randint(10000, 20000) for _ in range(5)]
T = sum(v * random.randint(0, 1) for v in vals)
print(f"Instance: {vals} {T}")

print((subsetsum(vals, T)))
```

Here's the idea; how many different inputs does `subsetsum` take? todo:2 to the n leaves? internal nodes are the same value?

4 Fibonacci numbers

Fibonacci numbers are a classic example of one of the first algorithms studied in the west. This is a recurrence relation defined by $F_0 = 0, F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$.

```
   def f(n):
    if n <= 1: return n
    return f(n-1) + f(n-2)
```

We write out a *giant* recursion tree just to compute $f(5)$, with 15 nodes, only increasing with each step. This number is getting very big: how long does this take? The number of steps $T(n) = T(n-1) + T(n-2) + 2$ has the same recurrence as F_n plus an additional factor 1, so $T(1) = 1, T(0) = 0$, and $T(n) \geq T(n-1) + T(n-2)$ which implies $T(n) \geq F_n$. How fast does F_n grow? We estimate it grows exponentially:

$$F_n \leq 2F_{n-1} \leq 4F_{n-2} \leq \cdots \leq 2^k F_{n-k} \leq \cdots \leq 2^{n-1}.$$

This is an upper bound. Is there a lower bound?

$$F_n \geq 2F_{n-2} \geq 4_{n-4} \geq 8F_{n-6} \geq \cdots \geq 2^k F_{n-2k} \geq \cdots \geq 2^{\frac{n}{2}-1} \approx 1.4^n.$$

So the Fibonacci numbers grow in between 1.4^n and 2^n. Therefore this is an exponential time algorithm, since the recursion tree is exponentially large. Our current algorithm is kind of dumb. We are doing lots of extra work.

```
memo = {}
def f2(n):
    if n in memo: return memo[n]
    if n <= 1: return n
    ans = f2(n-1) + f2(n-2)
    memo[n] = ans
    return ans
```

This is *much* faster because we only need to compute each value of n once, so it takes one addition to compute. We can rewrite this recursion tree in a different way where we reuse the nodes. There is a directed acyclic graph on the previous values; this computation graph has n vertices and $2n$ edges. The recursion time is the number of possible paths from n to the base case. However, the "memo-ized" recursion is the number of vertices times the time per vertex, which is one addition, giving us $O(n)$ time.

4.1 Dynamic programming

This is the idea of **dynamic programming**. We will see many more examples of this later. In dynamic programming, we solve recursive problems quickly by storing the answers to the subproblems. There are different ways we can do this:

- "Top-down": Recursion with memo-ization
- "Bottom-up DP": Compute the answers bottom to top iteratively

```
def fibdp(n):
fibs = [0, 1]
for i in range(2, n+1):
    fibs.append(fibs[i-1] + fibs[i-2])
return fibs[n]
```

This gets around the (non-theoretical) issue of not being able to call recursion too many times. We can also do what is called a "sliding window" DP, where we only store the last couple of recursions and discard the rest, helping with space issues.

```
def fibwindow(n):
a, b = 0, 1
for i in range(n):
    a, b = b, a + b
return a
```

This algorithm is $O(1)$ space and $O(n)$ time, while the previous algorithm is both $O(n)$ space and time. Can we do better than linear time?

4.2 Matrix exponentiation method

Each step the sliding window algorithm can be thought of as applying a matrix $\left[\begin{smallmatrix}0&1\\1&1\end{smallmatrix}\right]$ to a vector $\left[\begin{smallmatrix}a\\b\end{smallmatrix}\right]$. In essence, a, b starts at $\left[\begin{smallmatrix}F_0\\F_1\end{smallmatrix}\right]$ and ends at

$$\begin{bmatrix}F_n\\F_{n+1}\end{bmatrix}=\begin{bmatrix}0&1\\1&1\end{bmatrix}^n\cdot\begin{bmatrix}0\\1\end{bmatrix},\quad\begin{bmatrix}0&1\\1&1\end{bmatrix}^n=\begin{bmatrix}F_{n-1}&F_n\\F_n&F_{n+1}\end{bmatrix}.$$

Given A, how quickly can we compute A^n? We split the squares; $A^{13}=A^8A^4A$, and compute each by repeated squaring. So this results in $\log n$ matrix multiplies. Therefore we can compute F_n with $O(\log n)$ arithmetic operations. Numbers get really big; we have $1.4^n \le F_n \le 2^n$, $F_n = 2^{\Theta(n)}$ which implies $\log F_n = \Theta(n)$. So writing down F_n takes $\Theta(n)$ bits.

something, assume arithmetic is constant only if numbers are polynomial large, wordram model?

5 Dynamic programming

So far we've been doing recursion, which entails solving a problem by formulating it as a series of choices. For example, we pick one choice and recuse on the rest. For the n queens, we decide where to do on the first row, then decide where to go on the rest. The benefits is that this is easy to find an accurate solution for many problems. The problem is that this is usually exponential time.

Dynamic programming can be thought of as "smart recursion", or "recursion without repetition". Last times example was subsetsum. This tree in principle could have 2^n leaves, but we store answers to intermediate problems. We think of recursion graphs as a DAG.

5.1 Interval scheduling

Say that you're an airbnb host and many people are requesting different times to stay at your vacation home, and different people will pay you to stay at different times. We are given a set I of **intervals** consisting of a start time,

finish time, and value (s_i, f_i, w_i) and we want to find a disjoint subset of maximum weight. In other words, we want to find $S \subseteq I$ that doesn't overlap $(s_i, f_i) = \emptyset$) maximizing weight, or

$$\sum_{i \in S} w_i \geq \sum_{i \in G} w_i$$

for any $G \subseteq I$. How would we write a scheduling function Sched(I)? For now, our goal is the weight of S. Pick any interval i; then return the maximum (Sched($I \setminus i$), w_i + Sched($I \setminus \{i$ or any j overlaps $i\}$). How do we make this fast? Pick $i = 1$ each time. All recursive calls only ever consider suffixes of I, so the number of different inputs is $n + 1$.

5.2 Longest increasing subsequence

Given $A_1, A_2, \cdots, A_n$, we want to find $s_1 < s_2 < \cdots < s_k \in [n]$ with $A_{s_1} \leq A_{s_2} \leq \cdots \leq A_{s_k}$ of maximum weight. How do we do this? We claim this corresponds to the maximum weight path of a graph. We build a graph as following: let $f(i) :=$ LIS strating with (i, A_i).

6 Practice problems

Problem 1. *n local businesses free products, certain times, business i will be giving away free times only at time T_i. for each pair (i, j), the amount of time $t_{ij} > 0$ to walk from i to j (satisfying the triangle inequality), design an $O(n^2)$ DP algorithm to compute maximum number of businesses,*

Solution. how to do this? First solve this as a recursion. Cast the problem as a **series of choices**; the choices are a sequence of merchants/stalls visited. Possible if $T_i - T_j \geq t_{ij}$ (can make it in time).

```
def solve(choices):
    for each next choice i
    if valid (T_i - T_choices) geq t_{i,choices}
        solve(choice, t[i])
```

Another option is this; "do I show up to stall i"? How good is a set of choices? We need to know where we end up (last stall visited, hard to compare at different stalls) and the number of visits. Define $f(i) :=$ maximum number of stalls you can visit before reaching stall i's giveaway. The answer is $\max_j f(j) + 1$, such that $T_i - T_i \geq T_{ij}$ for every i. The reason being we came from somewhere, being j. Therefore, this is the **recurrence**:

```
f(i)=max_j f(j)+1, T_i-T_j\geq t_{ij}
```

This is the **base case**,

```
f(0):=0
```

where business 0 is located at ??, T_0. The **recurrence subproblem** is

```
f(i):= max number of stalls you can visit
before ending up at stall i's giveaway
```

which we solved with the first algorithm. This has space $O(n)$, time $O(n^2)$. ■

Solution. latex. choices = which word to break after? $f(i)$ = min sum of sequences of gaps before a linebreak after i. two solutions end at the same word; what is the cost? subproblem; given word best subproblem.

If we end at a given word, given that we line break after word, all that matters is the sum of squares before. Write down base case, recurrence, answer. Basecase: $f(0) = 0$ no words. Answer: $\min_i f(i)$, wordds $i+1, \cdots, n$ fit in the line. Recursive subproblem: $f(i) = \min_{j<i} f(j)$+ cost of $i+1, \cdots, j$ line. ■

7 Knapsack

A classic example of DP is robbing a store. You have a knapsack, and you can only carry so much weight. The store has n items, each with value v_i and weight w_i. You have capacity C. What is the max value subset of items to carry with total weight $\leq C$? We could also write this as $\max_{S\subseteq[n], \sum_{i\in S} w_i \leq c} v_i$. How do we write a recursive solution?

```
knapsack(items, C)
    w,r = items[-1]
    return max(knapsack(items[:-1],C),
    knapsack(items[:-1],C-w)+v)

#base cases
    if C < 0: return -infinity
    if len(items)=0: return 0
```

What does DP say we are doing? We should make this fast by memoizing. Then the time depends on the number of vertices of our directed acyclic graph (DAG), representing todo:?? two dimensional table

todo:sliding window?

8 Greedy Algorithms

Consider interval scheduling. What do we do for greed?

(1) Earliest finish=latest start

(2) Duration (shortest)

(3) Earliest start (first come first serve)

(4) Smallest number of intervals it intersects

(2) doesn't work (small interval intersecting two), (3) doesn't work (long interval at the beginning). It turns out (1) works and (4) doesn't. For (4), duplicate intervals mess it up, you trick the algorithm (pushing it toward the wrong choice) by duplicating intervals that intersect bad choices. So early finish works in all cases, with runtime $O(n \log n)$.[2] The point is that it's easy to convince yourself that algorithms work (particularly with greedy algorithms), so we need proofs of correctness.

So how do we go about proving correctness?

Proof. We use induction on n. The base case is $n = 0$, which returns [], which is correct. Let $n \geq 1$, $|I| = n$. By the inductive hypothesis, suppose `GreedySchedule`(I') is correct for every I, $|I'| \leq n-1$. For any t, define $I_t := \{(s,f) \leq I \mid s \geq t\}$. In (s_1, f_1)=the first to finish in I, `GreedySchedule`(I) = $[(s_1,f_1)]$+`GreedySchedule`(I_{f_1}). Then $|I_{f_1}| \leq n-1$ implies `GreedySchedule`(I_{f_1}) is correct in I_{f_1}.

[2] Details: first sort the f_i by earliest end time ($\log n$), then take first interval (s,f). Repeatedly remove the first interval (s',f') if $s' < f$.

Let S^* be a true solution for I, then define $S^* = (s_1^*, f_1^*), (s_2^*, f_2^*), \cdots, (s_{k^*}^*, f_{k^*}^*)$, where $s_1^* < f_1^* \le s_2^* < f_2^* \le \cdots \le s_{k^*}^* < f_{k^*}^*$. Then `GreedySchedule` returns from S some $k^* \le k$. The claim is that $k \le k^*$. We know $f_1 \le f_1^*$ by the definition of `GreedySchedule`. But $f_1^* \le s_2^*$ by definition, so I_{f_1} contains $\{(s_2^*, f_2^*), (s_3^*, f_3^*), \cdots, (s_{k^*}^*, f_{k^*}^*)\}$.

We want to show that k is big. We have $k = |\texttt{GreedySchedule}(I)| = 1 + |\texttt{GreedySchedule}(I_{f_1})| = 1 + |\mathrm{OPT}(I_{f_1})|$ by the induction hypothesis. So this is greater than $1 + A$, for A *any* disjoint subset of I_{f_1}. Take $A = \{(s_2^*, f_2^*), (s_3^*, f_3^*), \cdots, (s_{k^*}^*, f_{k^*}^*)\}$, so $\mathrm{OPT} \ge 1 + (k^* - 1) = k^*$. Therefore $k \ge k^*$. ⊠

Example 8.1. Let us discuss another example. Consider files with length $L_1, L_2, \cdots, L_n$. If placed in order $\pi_1, \pi_2, \cdots, \pi_n$[3] (permutations of $[n]$), we have $\mathrm{cost}(k) = \sum_{i:\, \pi_i \le \pi_k} L_i$. Our goal is to define the layout π minimizing the average $\frac{1}{n}\sum_{k=1}^{n} \mathrm{cost}(k)$. Every file access has to access the first one, so you want to make sure the earliest ones are smallest. How do we prove this?

Suppose we exist we have some two out of order.

swapping → bubble sort. know bubble sort gets to a better solution, which implies quicksort also gets a better solution.

9 Stable marriage

We want to match n students and n positions. Each student/position has a ranked order of preference. Replace students/positions with men/women and we get **stable marriage**, where the matching is "stable"; we need partners to prefer their matches. The goal is the following: given preference lists, find a stable matching.

Example 9.1. Let $\mathrm{Men} = \{A, B, C\}$ and $\mathrm{Women} = \{X, Y, Z\}$. Here A prefers $Y > X > Z$, B prefers $Y > X > Z$, C prefers $X > Y > Z$. Unanimously the women prefer $A > B > C$. The pairing is then $(A, Y), (B, X), (C, Z)$.

Another example. The preferences are:

(A) $X > Z > Y$, (X): $B > A > C$

(B) $Z > X > Y$, (Y): $C > A > B$

(C) $X > Y > Z$, (Z): $A > B > C$

First you form the initial assignments, assigning $(A, X), (B, Y), (C, Z)$. This is not stable. (B, Y) will break up, (C, Z) will break up, A prefers Z, ?? follow the tree, four breakups will go back to where we start.

It is not obvious that any stable solution exists, and it is quite surprising that we can prove that one exists. This is proven by the algorithm we will give.

9.1 Gale-Shapley Stable Marriage Algorithm

We start with nobody matched. We repeat the following process: Let $\mathrm{Men} = \{A_i\}$, $\mathrm{Women} = \{X_i\}$ for $i \in \{1, 2, \cdots, n\}$. For $n = 3$, pick any unattached man. If each man A_i has preference $X_{\sigma_i(1)} > X_{\sigma_i(2)} > \cdots$ for some permutation $\sigma_i \in S_n$, A_i will ask $X_{\sigma_i(1)}$ on a date. If $X_{\sigma_i(1)}$ is unattached, she will accept (or if she prefers to her current partner). If the man is rejected, the man crosses her off his list (so if we return to A_i, the next candidate will be $X_{\sigma_i(2)}$). Repeat forever. We go down to the bottom of the list, where someone will eventually accept. So this matches everybody.

Theorem 9.1. *The Gale-Shapley Stable Marriage Algorithm returns a stable matching in order $O(n^2)$ time.*

[3] ***Not*** the homotopy groups!

Proof. We need to show that this algorithm terminates first. To do this, we need to make progress every step of the way. In each round, either a man crosses a woman off, or a woman moves up the ladder (improved happiness by rejection). We never uncross the list, and women never get traded down. The maximum number of things we can cross off is n^2 (men times women), and you can do n^2 improvements per woman. Overall, there are only $O(n^2)$ rounds. This shows that the algorithm terminates, and the final solution is indeed a matching (if there wasn't, repeated with the unattached man).

The question is, is this matching stable? Suppose the matching is not stable. That means there exist (A, B, X, Y) such that (A, X), (B, Y) are matched but (WLOG) (A, Y) would elope. This means $Y > X$ on A's list, and $A > B$ on Y's list. How did A get mapped to X? This implies A proposed to X at some point, which means X is at the top of A's list. This can only happen if Y crossed A off. So at some point, A proposed to Y and was rejected. Why? Because at some point, Y was rejected by some $C > A$. This is a problem because no trading down implies that the final match $B > A$ on Y's list, contradicting the assumption. Therefore the marriage is stable. ⊠

Example 9.2. Consider the pairing

(A) $X > Y > Z$, (X): $C > B > A$

(B) $Y > Z > X$, (Y): $A > C > B$

(C) $Z > X > Y$, (Z): $C > B > A$

People end up bery unhappy. ?? Here Gale-Shapley is optimal for the men (proposers) and pessimal for women (receivers). That is to say, let S_A denote the set of possible matches for A in any stable assignment. Then best(A) is the highest ranked in S_A, analogously worst(A) is the lowest ranked in S_A. Gale-Shapley then sends $A \to$ best(A) for every man A and sends $B \to$ worst(B) for every woman B. This is sort of surprising, let us prove it.

Lemma 9.1. *Each man is only rejected by women who cannot match them in any stable matching.*

Proof. We use induction on the number of steps taken (rounds). After 0 steps, this is true because no man has been rejected by anybody. We know by induction on previous rounds that this is true on the $(k-1)$th round. In a given round, if a man A is rejected by a woman X, that means she prefers B. This implies that B has X at the top of his list (everybody above X is crossed off). By induction, these rejections resulted in unstable matchings. This implies in any stable matching, B matched to X or lower on his list. So in any stable matching, X is matched to B or higher on her list. This means $A < B$ or (X, A) are not matched. Each man then gets the best possible outcome, or $A \to$ best(A) for every man A. ⊠