# Notes on Algorithms

Jennifer Zajac

Summer 2 2021

# Contents

# List of Tips

# Chapter 1

# Algorithms

## 1.1 Overview

**algorithm** An explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions for solving a computational problem.

**algorithms** The study of how to solve computational problems.

**computational problem** Can be solved by computer; has a precise, well-defined, correct answer.

# Part I

# Problems

# Chapter 2

# Counting

**Counting function**

**Input:** A collection of objects.
**Problem:** Determine how many objects are in the collection.
**Output:** The number of objects (positive integer).

## 2.1 Simple Counting

**Simple Counting with Students**

1: Start count at 0 and pass ball to a student
2: **repeat**
3: | Student with ball sets their number to (1 + what last person said)
4: | Student with ball says their number Student with ball passes to a standing student & sits
5: **until** only 1 student is standing
6:
7: Student with ball sets their number to (1 + what last person said)
8: Student with ball says their number

**Explanation:** Each student counts for 1. Loop invariant: after each iteration, sum of standing students' numbers = total number of students. Every student is always accounted for.

**loop invariant** After every iteration of a loop, this property is true.

## 2.2 Recursive Counting

**Recursive Counting with Students**

1: Everyone set your number to 1
2: **repeat**
3: | Everyone partner up, wait if none found
4: | Set your number to (your number + partner's number), if waiting keep same number
5:
6: | One partner sits down
7: **until** only 1 student is standing
8:
9: The standing student says "the total is (their number)"

**Observations:** when there are $n$ students, and each line counts as 1 step...

- $T(1) = 2$.
- If $n > 1$, after 1 loop there are $\lceil n/2 \rceil$ students left. The ceiling considers the case of an odd student.
- $T(n) = 3 + T(\lceil n/2 \rceil)$, where there are 3 steps this loop and $T(\lceil n/2 \rceil)$ steps total in the following loops.
- $T(2^m) = 3m + 2$. Students get halved $m$ times.
  **Derivation:**

$$
\begin{aligned}
& 3 + T(\lceil 2^m/2 \rceil) && \text{Plugging in } 2^m. \\
&= 3 + T(\lceil 2^{m-1} \rceil) && 2^m \text{ is an integer. Ceiling is redundant.} \\
&= 3 + T(2^{m-1}) && \\
&= 3 + 3 + T(2^{m-2}) && \\
&= 3 + 3 + 3 + T(2^{m-3}) && \text{For each loop, add } +3 \text{ and subtract 1 from } m. \\
&= 3k + T(2^{m-k}) && \text{After } k \text{ substitutions.} \\
&= 3 + ... + 3 + T(2^{m-m}) && \text{After m substitutions, there are } m \text{ 3's.} \\
&= 3 + ... + 3 + T(1) && T(1) \text{ is the base case.} \\
&= 3m + 2. &&
\end{aligned}
$$

  See proof by induction.

- $T(n) = 3 \log_2 n + 2$.
  **Derivation:** If $T(2^m) = 3m+2$ for every number of students $n = 2^m, \log_2 n = m$. Substituting $m$ in, $T(n) = 3 \log_2 n + 2$.

**Ceiling is redundant for integers.** You can drop the ceiling operation around an integer.

### 2.2.1 Prove $T(n) = 3m + 2, \ \forall n = 2^m$

**Claim:** Recurrence $T(1) = 2, T(n) = 3 + T(\lceil n/2 \rceil)$ has closed form $T(n) = 3m + 2$ for $\forall n \in \mathbb{N}, n = 2^m$.

*Proof by induction.* Let $H(m)$ be the statement $T(2^m) = 3m + 2$.
**Base:** LHS: $H(0) = T(2^0) = T(1) = 2$. RHS: $3(0) + 2 = 2$. LHS = RHS, so $H(0)$ is true.

**Inductive step:** Assume $H(m-1)$ is true to show $H(m)$ is true, that $T(2^m) = 3m + 2$.
Since $H(m-1)$ is true, $T(2^m - 1) = 3(m-1) + 2$ by inductive hypothesis.

$$
\begin{aligned}
T(2^m) &= 3 + T(\lceil (2^m/2) \rceil) \\
&= 3 + T(\lceil 2^m - 1 \rceil) \\
&= 3 + T(2^m - 1) && \text{Integer doesn't need ceiling.} \\
&= 3 + 3(m-1) + 2 && \text{By inductive hypothesis.} \\
&= 3 + 3m - 3 + 2 \\
&= 3m + 2
\end{aligned}
$$

$\therefore H(m)$ is true. □

## 2.3 Comparison

- **Simple counting:** $T(n) = 3n$ steps
- **Recursive counting:** $T(n) = 3 \log_2 n + 2$ steps.

**Best:** Recursive counting.

# Chapter 3

# Stable Matching

**Stable Matching function**

Assume there are an equal number $n$ of doctors and hospitals.
**Input:**

- $n$ doctors: $d_1 ... d_n$
- $n$ hospitals: $h_1 ... h_n$
- each doctor's ranking of hospitals (ex. $d_1 : h_2 > h_3 > h_1$)
- each hospital's ranking of doctors (ex. $h_1 : d_1 > d_3 > d_2$)

**Problem:** Create a stable assignment of doctors to hospitals, so no one switches jobs.
**Output:** The stable matching of doctors and hospitals.

**matching** $M$ A non-empty set of doctor-hospital pairs where no doctor/hospital appears twice.

**perfect matching** Every doctor/hospital appears once.

Terminology:

- "d is matched to h": $(d, h) \in M$
- "d is matched": $(d, h) \in M$ for some $h$

**unstable matching** Some doctor-hospital pair prefer one another to their mate in the matching.

Instabilities:

1. A matched doctor prefers an unmatched hospital. $d, h$ such that $d$ is matched to $h'$, $h$ is unmatched, but $d : h > h'$.
2. A matched hospital prefers an unmatched doctor. $d, h$ such that $h$ is matched to $d'$, $d$ is unmatched, but $h : d > d'$.
3. Two pairs want to swap partners. $d, h$ such that $d$ is matched to $h'$, $h$ is matched to $d'$, but $d : h > h'$ and $h : d > d'$.

Note that #1 and #2 cannot happen in a perfect match.

**stable matching** A perfect matching that has no instabilities.

There can be multiple stable matchings. A doctor/hospital may be with their 1st pick in one, and their 5th pick in another.

When there are different numbers of doctors and hospitals, the unmatched doctors/hospitals are still stable because no current pairs want to switch.

If all hospitals wanted the same doctor, assign that doctor to their 1st pick. This is never unstable, because that doctor would not want to leave.

## 3.1 Gale-Shapley Algorithm

**Gale-Shapley for Stable Matching**

```
1:  Let M be empty
2:  while some hospital h is unmatched do
3:      if h has offered a job to everyone then
4:          break                ▷ Return matchings. Does not apply if same number doctors/hospitals
5:
6:          ▷ JOB OFFER ◁
7:      else let d be the highest-ranked doctor to which h has not yet offered a job
8:          h makes an offer to d
9:          if d is unmatched then
10:             d accepts, add (d,h) to M
11:         else if d is matched to h' & d: h' > h then
12:             d rejects, do nothing                ▷ d prefers their current match
13:         else if d is matched to h' & d: h > h') then
14:             d accepts, remove (d,h') from M and add (d,h) to M   ▷ d dislikes their current match
15:
16: Output M
```

**The party making offers (based in order of their preferences) get a stable matching optimal for them.** This is optimal for the hospitals (they get their best pick). (pareto-optimal). If doctors make offers to hospitals, results in a different stable matching that is optimal for doctors.

Assumes no ties in individual rankings.

Observations:

- **Hospitals make offers in descending order.** Hospitals make offers to doctors that they prefer less and less. Hospitals' happiness decreases during algorithm.
- **Doctors that get a job never become unemployed.** Once a doctor is matched, they never become unmatched with a hospital. They would only leave a hospital for another hospital. If a doctor is matched at some point, they are matched forever. Even if there were more doctors than hospitals, doctors never leave to become unemployed.
- **Doctors accept offers in ascending order.** Doctors receive offers that they prefer more and more. Doctors' happiness increase during algorithm.

### 3.1.1 Terminates after at most $n^2$ job offers

**Claim:** The GS algorithm terminates after at most $n^2$ iterations of the main loop ($n^2 joboffers$). Loop stops when a hospital has made an offer to every doctor and was rejected.

Each hospital can offer each doctor at most 1 time = unique offers happen once. $n$ doctors $*$ $n$ hospitals = At most, there are $n^2$ unique offers.

### 3.1.2 Creates a perfect matching

**Claim:** The GS algorithm returns a perfect matching (all doctors/hospitals are matched).
*Assume equal number of doctors and hospitals, or perfect matching is impossible.*

*Proof by contradiction.* Suppose the claim is false: the Gale-Shapley algorithm does not return a perfect matching. Then there is some doctor $d$ which is not matched and also some hospital $h$ which is left out. $h$ is unmatched at the end of the algorithm, so $h$ has offered a job to everyone (included doctor $d$). So $h$ offered a job to $d$.

- Case 1: doctor $d$ accepts offer. By observation 2, doctors stay employed, so $d$ has to be matched at the end of the algorithm. Contradiction: $d$ cannot be unmatched, but is.
- Case 2: doctor $d$ rejects offer. Doctors only reject if they are already paired, and by observation 2, doctors stay employed. So doctor $d$ cannot be unmatched at the end. Contradiction: $d$ cannot be unmatched, but is.

□

### 3.1.3 Creates a stable matching

**Claim:** The GS algorithm returns a stable matching.

*Proof by contradiction.* Suppose the claim is false: the Gale-Shapley algorithm does not return a stable matching. Then there is an instability in the matching $M$: $(d, h'), (d', h)$, such that $h : d > d'$ and $d : h > h'$. Since $h$ prefers $d$, we know $h$ made an offer to $d$ before $d'$.

- Case 1: $d$ rejected the offer. Then we know $d$ prefers their current hospital $h*$ to hospital $h$, that $d : h* > h$. Putting this together, $d : h* > h > h'$. By observation 3, doctors get happier as algorithm progresses. If doctor $d$ ended up with $h'$, then $d$ must prefer $h'$ over $h*$, so $d : h' > h* > h$. Contradiction: $d : h* > h > h' \nrightarrow\!\!\!\!\!\leftarrow d : h' > h* > h$.
- Case 2: $d$ accepted the offer. $(d, h) \in M$ at some point. By observation 3, doctors get happier. Since $d$ ended up with $h'$, then $d : h' > h$. This contradicts $d : h > h'$. Contradiction: $d : h' > h \nrightarrow\!\!\!\!\!\leftarrow d : h > h'$.

□

### 3.1.4 Total number of operations

- Doctors' preferences stored as ranked list of hospitals: In the worst case, uses $n^3$ operations. Loop runs $\leq n^2$ times $= n^2$ job offers. Each offer uses $\leq n$ operations: $h$ has to be found in the doctor's list of preferences.
- Doctors' preferences stored as list of rankings indexed by hospital: In the worst case, uses $n^2$ operations. Each offer uses 1 operation since the position in the list is known.

# Part II

# Proof techniques

# Chapter 4

# Proof by Induction

Used to prove a claim $H$ is true for every natural number $i$ starting at a first value (usually 0 or 1). $H(i)$ is true $\forall i$.
Steps:

1. Base case: prove directly (by plugging in the base cases).
2. Inductive step: for a general $k \in \mathbb{N}$, show $H(k-1) \implies H(k)$. Use the **inductive hypothesis (IH)**: assume $H(k-1)$ is true. Alternatively, show $H(k1) \implies H(k+2)$ and assume $H(k)$ is true.
    - Strong induction: We may assume $H(1), H(2), ... H(i-1)$ to prove $H(i)$ Some number in this chain will be helpful (excluding $k-1$). Cases may be helpful, but not necessary.
    - Weak induction: We only need to assume $H(i-1)$ because it is all we need to prove $H(i)$. Anything proven with weak induction can be proven with strong induction. Weak induction is like having all previous steps proven, but only using the $(k-1)$ step.

Explanation:

- Starting from the base, $H(1) \implies H(2)$ by inductive step $\implies H(3) \implies ... \implies H(100) \implies ...$, to any natural number!

Note that induction can be used to prove algorithms, and does not only apply to the following examples.

### 4.0.1 Summation

**Claim:** For every $n \geq 1, \sum_{i=0}^{n-1} 2^i = 2^n - 1$.

*Proof by Induction.* **Base:** $n = 1$. LHS: $\sum_{i=0}^{i=0} 2^i = 2^0 = 1$. RHS: $2^1 - 1 = 1$. LHS = RHS. $H(1)$ is true.

**Inductive step:** Assume $H(n)$ holds. Show $H(n+1)$, that $\sum_{i=0}^{n+1-1} 2^i = 2^{n+1} - 1$.

$$\begin{aligned}
&\sum_{i=0}^{n+1-1} 2^i \\
&= \sum_{i=0}^{i=n} 2^i \\
&= 2^0 + 2^1 + 2^2 + ... + 2^n && \text{The } n-1 \text{ terms are in IH.} \\
&= 2^n - 1 + 2^n && \text{By induction hypothesis.} \\
&= 2 * 2^n - 1 \\
&= 2^{n+1} - 1
\end{aligned}$$

**Alternatively,** $H(k-1) \implies H(k)$.
Assume $\sum_{i=0}^{i=k-1-1} 2^i = 2^{k-1} - 1 = \sum_{i=0}^{i=k-2} 2^i = 2^{k-1} - 1$ is true. Show $H(k) : \sum_{i=0}^{i=k-1} 2^i = 2^n - 1$.
Pull the last term out of the summation.

$$\begin{aligned}
&\sum_{i=0}^{i=k-2} 2^i + \sum_{i=k-1}^{i=k-1} 2^i \\
&= \sum_{i=0}^{i=k-2} 2^i + 2^{k-1} \\
&= 2^{k-1} - 1 + 2^{k-1} && \text{By induction hypothesis.} \\
&= 2 * 2^{k-1} - 1 \\
&= 2^k - 1
\end{aligned}$$

□

### 4.0.2 Prime factors

**Claim:** $\forall n \in \mathbb{N}, n \geq 2$ has at least 1 prime factor.

> **prime** Positive integer greater than 1 that cannot be formed by multiplying two smaller natural numbers.
>
> **factor** A number that divides another number evenly.

We need to use strong induction. Weak induction won't work: 11 having a prime factor doesn't help determine if 12 has a prime factor.

*Proof by Induction with Cases.* **Base:** 2 is a natural number that has the prime factor 2.
**Inductive step:** Assume $H(2), H(3), ..., H(k-1)$ are all true. Show $H(k)$ is true, that $k$ has a prime factor.
Proof by cases:

- Case 1: $k$ is prime. $k$ is a factor of itself ($k * 1 = k$), so $k$ is a prime factor of $k$.
- Case 2: $k$ is not prime, so $k$ is divisible by a natural number less than $k$. $k = a * b$ where $a, b \in \mathbb{N}$ and $2 \leq a, b < k$. By IH, all natural numbers less than $k$ have a prime factor, so $H(a)$ is true (we don't know where $a$ is in the chain, but it's somewhere in there). Therefore, $a$ has a prime factor: $a = x * y$ where $x$ is prime. $k = a * b = x * y * b$, where $x$ is prime and $y * b$ is a natural number. Therefore, $x$ must be a prime factor of $k$.

□

# Chapter 5

# Proof by Contradiction

Explanation:

- No claim can be both true and false.

Steps:

1. Assume the claim $H$ is false ($\not{H}$ is true).
2. Show $H$ being false implies contradictory assertions (that both assertion $Q$ and $\not{Q}$ are true)
3. $Q$ and $\not{Q}$ cannot both be true, so $H$ must not be false ($H$ is true).

# Part III

# Algorithm techniques

# Part IV

# Analysis

# Chapter 6

# Asymptotic Analysis

How does running time grow as the size of the input grows?

- Tool used to compare algorithms by asymptotic order of growth, the behavior as $n \implies \infty$. # operations $\rightarrow$ runtimes $\rightarrow$ functions. Order of growth functions are easier to work with than the exact running time/# of operations
- Describes performance based on input size $n$.
- Measures speed/size as input grows very big to see how algorithm scales. Focuses on the dominant (largest) term of function. Generally we don't care about small inputs, because then all algorithms will be reasonably quick.

## 6.1 "Big-Oh" Notation

$f(n) = O(g(n))$ if there exists $c \in (0, \infty)$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq c * g(n)$ for every $n \geq n_0$.

- $f(n)$ is exact run time based on # of operations. $g(n)$ is order of growth.
- Asymptotic version of $f(n) \leq g(n)$.
- Roughly equivalent to $\lim n \rightarrow \infty f(n)/g(n) < \infty$. Denominator grows faster = converges. Can solve using L'Hopitals.
- **There are many combos of $c$ and $n_0$ that will work in the inequality.**
- $f(n)$ can be Big Oh of many functions, starting from the order of growth.

  (Big Theta is the explicit order of growth. g(n) has the EXACT same order of growth as f(n), but there are an infinite number of functions that have the same order of growth.)

### 6.1.1 One easy way to find $c$ and $n_0$

1. To get $c$, sum the coefficients in $f(n)$.
2. $n_0 = 1$.