## CSE 417 Autumn 2025

# Lecture 5: Running Time

Nathan Brunelle

# Homeworks

HW 1 due today at 11:59pm.

HW 2 out today at 11:30am.

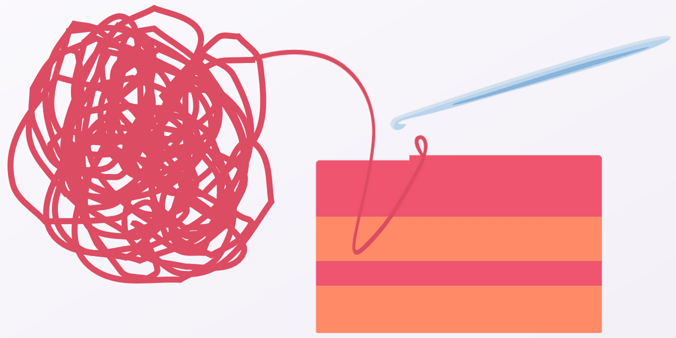
# Motivating Example

# Let's design an algorithm

- I have a pile of string
- I have one end of the string in-hand
- I need to find the other end in the pile
- How can I do this efficiently?

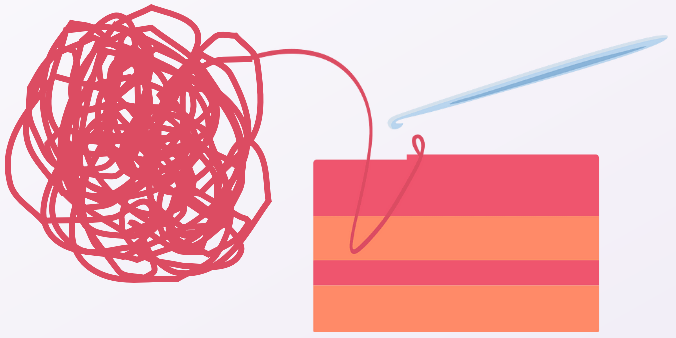
# Algorithm Ideas

Whatcha got?



# My Approach

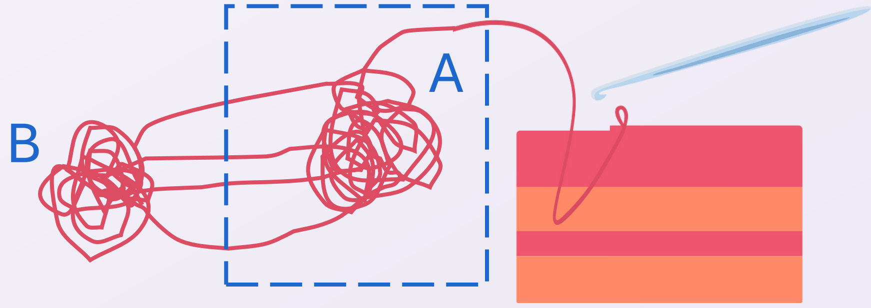
# End-of-Yarn Finding Algorithm



Set aside the already-obtained beginning

Do the following until you find the end:

- Separate the pile of yarn into 2 piles
- Label A to be the pile that the beginning enters
- Label B to be the other pile
- Count the number of strands crossing the piles
- If count is even, set the pile to be A
- Otherwise set the pile to be B.



# Resource Analysis

# Why do resource analysis?

Allows us to compare *algorithms*, not implementations

- Using observations necessarily couples the algorithm with its implementation
- If my implementation on my computer takes more time than your implementation on your computer, we cannot conclude your algorithm is better

We can predict an algorithm's running time before implementing

Understand where the bottlenecks are in our algorithm

# Process for resource Analysis

End Result: A *function* which maps the algorithm's input size to count of resources used

Input of the function: **sizes** of the input

- Number of characters in a string, number of items in a list, number of pixels in an image

Output of the function: **counts** of resources used

- Number of times the algorithm adds two numbers together, number times the algorithm does a > or < comparison, maximum number of bytes of memory the algorithm uses at any time

Important note: Make sure you know the "units" of your input and output!

# Resource Analysis – Worst Case Running Time

If an algorithm has a worst case running time of $f(n)$

- Among all possible size-$n$ inputs, the "worst" one will do $f(n)$ "operations"

- I.e. $f(n)$ gives the maximum operation count from among all inputs of size $n$

# Resource Analysis – Best Case Running Time

If an algorithm has a **best** case running time of $f(n)$

- Among all possible size-$n$ inputs, the “**best**” one will do $f(n)$ “operations”

- I.e. $f(n)$ gives the **minimum** operation count from among all inputs of size $n$

# Resource Analysis – Worst Case Space

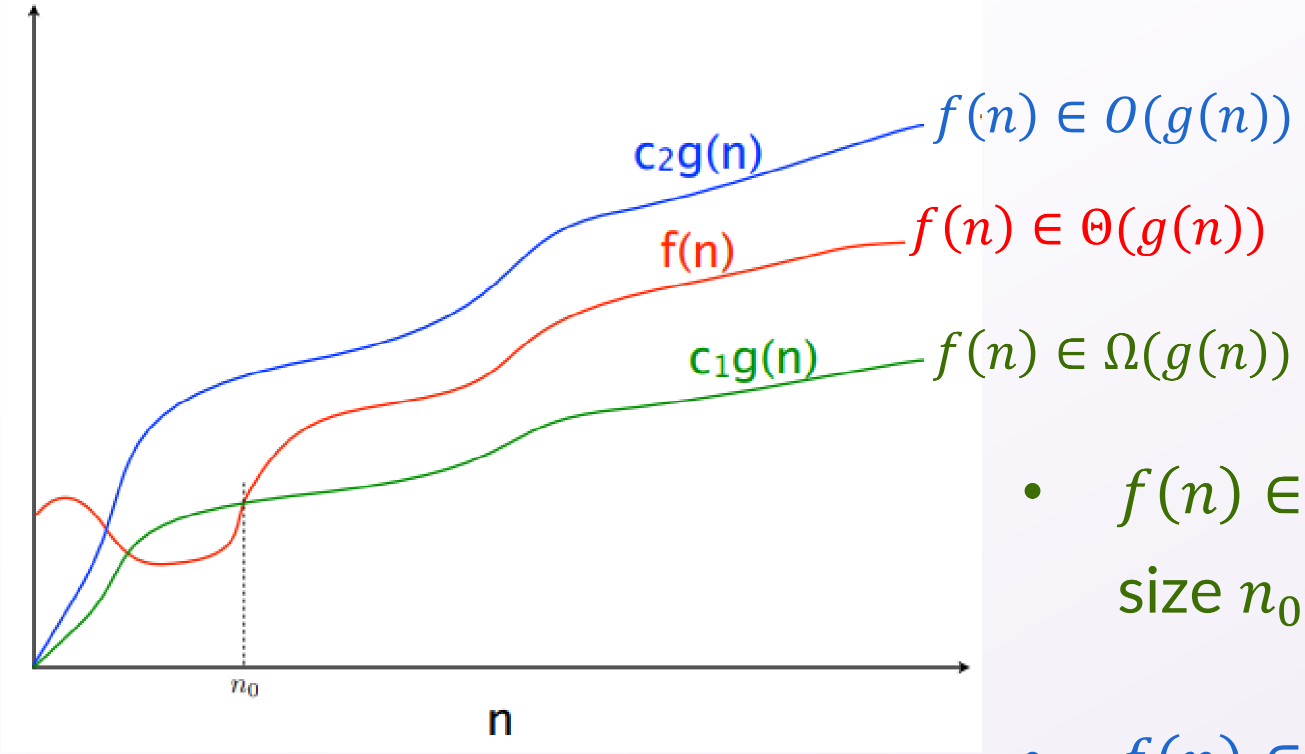
If an algorithm has a worst case space of $f(n)$

- Among all possible size-$n$ inputs, the “worst” one will use $f(n)$ bits of memory

- I.e. $f(n)$ gives the maximum amount of memory required from among all inputs of size $n$

# Asymptotic Notation

# Asymptotic Notation – Comparing Functions




- $f(n) \in \Omega\big(g(n)\big)$ provided that after some input size $n_0$, $f(n) \geq c_1 \cdot g(n)$ for some constant $c_1$
- $f(n) \in O\big(g(n)\big)$ provided that after some input size $n_0$, $f(n) \leq c_2 \cdot g(n)$ for some constant $c_2$
- $f(n) \in \Theta\big(g(n)\big)$ provided $f(n) \in O\big(g(n)\big)$ and $f(n) \in \Omega\big(g(n)\big)$

# Asymptotic Notation Examples

- $10n + 100 \in O(n^2)$
  - After $n_0 = ??$ with $c = ??$ we have $10n + 100 \leq cn^2$
- $13n^2 - 50n \in \Omega(n^2)$
  - After $n_0 = ??$ with $c = ??$ we have $13n^2 - 50n \geq cn^2$
- $n^2 + 3n \in O(4n^3)$
  - After $n_0 = ??$ with $c = ??$ we have $n^2 + 3n \leq c4n^3$

# Asymptotic Notation Examples

- $10n + 100 \in O(n^2)$
  - After $n_0 = 101$ with $c = 1$ we have $10n + 100 \leq cn^2$
- $13n^2 - 50n \in \Omega(n^2)$
  - After $n_0 = 100$ with $c = \frac{1}{2}$ we have $13n^2 - 50n \geq cn^2$
- $n^2 + 3n \in O(4n^3)$
  - After $n_0 = 1$ with $c = 1$ we have $n^2 + 3n \leq c4n^3$

# Running Time Example: Selection Sort

# Reminder: Selection sort

**Input:** Array $A[1 \dots n]$ of numbers

**Goal:** A permutation of $A$ that is sorted in decreasing order

1. **for** $i = 1, \dots, n$ **do**
2. Let $A[j]$ be the maximum element of $A[i \dots n]$.
3. Swap $A[i]$ and $A[j]$.
4. **return** $A$

**What should our input size units be?**

**What operations should we count?**

# Selecting Running Time Units

Input size units:

- Represents the size of the input
- You typically want this to in discrete intervals (i.e. the size should always be an integer)
    - E.g. Number of elements in a data structure, number of indices in an array, number of characters in a string, bit in a number

Running time operations:

- Count these to express running time
- Ideally these will have the properties of:
    - *Necessity* – All algorithms solving this type of problem will do this operation
    - *Frequency* – This operation is not at least as often as any other operation we might want
    - *Magnitude* – This operation is expensive to perform

# Selection sort Units

**Input:** Array $A[1 \dots n]$ of numbers

**Goal:** A permutation of $A$ that is sorted in decreasing order

1. **for** $i = 1, \dots, n$ **do**
2. Let $A[j]$ be the maximum element of $A[i \dots n]$.
3. Swap $A[i]$ and $A[j]$.
4. **return** $A$

**What should our input size units be?** Length of $A$

**What operations should we count?** Number of Comparisons

# Selection sort – Worst Case Running Time

**Input:** Array $A[1 \dots n]$ of numbers

**Goal:** A permutation of $A$ that is sorted in decreasing order

1. **for** $i = 1, \dots, n$ **do**
2. Let $A[j]$ be the maximum element of $A[i \dots n]$.
3. Swap $A[i]$ and $A[j]$.
4. **return** $A$

**Describe the inputs that cause the most comparisons.**

**In this case, all are equal!**

**How many are done?** $O(n^2)$

# Running Time Example: Gale-Shapley

# Reminder: Gale–Shapley algorithm

1. **while** there is a free proposer $p \in P$ **do**
2.     Let $r$ be the top remaining person on $p$'s preference list.
3.     **if** $r$ is also free **then**
4.         Have $r$ accept $p$.
5.     **else if** $r$ is paired but prefers the new proposer $p$ **then**
6.         Have $r$ accept $p$ and reject their current match $p'$.
7.     **else** (if $r$ is paired and prefers their current match)
8.         Have $r$ reject $p$.
9. **return** all matches

**What should our input size units be?** Size of $P$

**What operations should we count?** ???

# What operations do we need?

1. **while** there is a free proposer $p \in P$ **do**
2. Let $r$ be the top remaining person on $p$'s preference list.
3. **if** $r$ is also free **then**
4. Have $r$ accept $p$.
5. **else if** $r$ is paired but prefers the new proposer $p$ **then**
6. Have $r$ accept $p$ and reject their current match $p'$.
7. **else** (if $r$ is paired and prefers their current match)
8. Have $r$ reject $p$.
9. **return** all matches

Things we need to do:

- Iterate over free proposers
- Check if someone is matched
- Look up someone's current match
- Compare preferences
- Make/unmake a match

# Operations to count

Comparing “agents” (proposers with receivers, proposers with proposers, receivers with receivers)

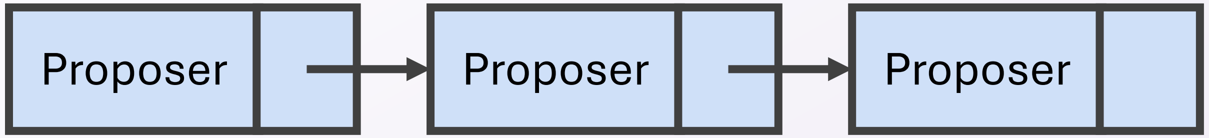
This includes:

- Equality checks (e.g. is this the receiver we’re looking for?)
- Matched queries (e.g. is this receiver matched, if so then to whom?)
- Preference checks (e.g. does this receiver prefer proposer 1 or 2?)

# Data Structures – first attempt

**Linked list for free proposers**

- Originally contains all proposers
- Match the first on in the list, then remove
- Re-add to the end if unmatched




**Two 1d arrays for matches**

- One where index $i$ has the proposer matched with receiver $i$, $-1$ if unmatched
- One where index $i$ has the receiver matched with proposer $i$, $-1$ if unmatched

| | | | |
|---|---|---|---|
| matched with $p_0$ | matched with $p_1$ | matched with $p_2$ | matched with $p_3$ |

| | | | |
|---|---|---|---|
| matched with $p_0$ | matched with $p_1$ | matched with $p_2$ | matched with $p_3$ |

**Two 2d arrays for preferences**

- One where index $i, j$ has the $j$th favorite proposer for receiver $i$
- One where index $i, j$ has the $j$th favorite receiver for proposer $i$

| | | | |
|---|---|---|---|
| First favorite of $p_0$ | Second favorite of $p_0$ | Third favorite of $p_0$ | Fourth favorite of $p_0$ |
| First favorite of $p_1$ | Second favorite of $p_1$ | Third favorite of $p_1$ | Fourth favorite of $p_1$ |
| First favorite of $p_2$ | Second favorite of $p_2$ | Third favorite of $p_2$ | Fourth favorite of $p_2$ |
| First favorite of $p_3$ | Second favorite of $p_3$ | Third favorite of $p_3$ | Fourth favorite of $p_3$ |

**One 1d array for proposals**

- Index $i$ has the last receiver that proposer $i$ has proposed to.

| | | | |
|---|---|---|---|
| $p_0$'s last proposal | $p_1$'s last proposal | $p_2$'s last proposal | $p_3$'s last proposal |

# Running time of each step

**while** there is a free proposer $\boldsymbol{p} \in \boldsymbol{P}$ **do**

    Let $\boldsymbol{r}$ be the top remaining person on $\boldsymbol{p}$'s preference list. $O(1)$

    **if** $\boldsymbol{r}$ is also free **then** $O(1)$

        Have $\boldsymbol{r}$ accept $\boldsymbol{p}$. $O(1)$

    **else if** $\boldsymbol{r}$ is paired but prefers the new proposer $\boldsymbol{p}$ **then** $O(n)$

        Have $\boldsymbol{r}$ accept $\boldsymbol{p}$ and reject their current match $\boldsymbol{p}'$. $O(1)$

    **else** (if $\boldsymbol{r}$ is paired and prefers their current match) $O(n)$

        Have $\boldsymbol{r}$ reject $\boldsymbol{p}$. $O(1)$

**return** all matches

$O(n^2)$

Overall: $O(n^3)$

What is the bottleneck?

Where should we focus if we want to make it faster?

# Data Structures – Better

| | | | |
|---|---|---|---|
| $p_0$'s preference for $r_0$ | $p_0$'s preference for $r_1$ | $p_0$'s preference for $r_2$ | $p_0$'s preference for $r_3$ |
| $p_1$'s preference for $r_0$ | $p_1$'s preference for $r_1$ | $p_1$'s preference for $r_2$ | $p_1$'s preference for $r_3$ |
| $p_2$'s preference for $r_0$ | $p_2$'s preference for $r_1$ | $p_2$'s preference for $r_2$ | $p_2$'s preference for $r_3$ |
| $p_3$'s preference for $r_0$ | $p_3$'s preference for $r_1$ | $p_3$'s preference for $r_2$ | $p_3$'s preference for $r_3$ |

**Linked list for free proposers**

- Originally contains all proposers
- Match the first on in the list, then remove
- Re-add to the end if unmatched

**Two 1d arrays for matches**

- One where index $i$ has the proposer matched with receiver $i$, $-1$ if unmatched
- One where index $i$ has the receiver matched with proposer $i$, $-1$ if unmatched

**Two 2d arrays for preferences**

- ~~One where index $i, j$ has the $j$th favorite proposer for receiver $i$~~
- ~~One where index $i, j$ has the $j$th favorite receiver for proposer $i$~~
- One where index $i, j$ has the index of receiver $j$ in proposer $i$'s preference list
- One where index $i, j$ has the index of proposer $j$ in receiver $i$'s preference list

**One 1d array for proposals**

- Index $i$ has the last receiver that proposer $i$ has proposed to.

# Improved running time of each step

**while** there is a free proposer $p \in P$ **do**
    Let $r$ be the top remaining person on $p$'s preference list. $O(1)$
    **if** $r$ is also free **then** $O(1)$
        Have $r$ accept $p$. $O(1)$
    **else if** $r$ is paired but prefers the new proposer $p$ **then** $O(1)$
        Have $r$ accept $p$ and reject their current match $p'$. $O(1)$
    **else** (if $r$ is paired and prefers their current match) $O(1)$
        Have $r$ reject $p$. $O(1)$
**return** all matches

$O(n^2)$

Overall: $O(n^2)$