THE MOST BASIC DATA STRUCTURES:

ARRAYS AND LISTS:

• ARRAYS:

A

1 2 i n

• WHAT IS THE i-th ELEMENT OF THE LIST?

x = A[i], O(1)

• DOES x belong to A?

In general case: O(n)

If A is sorted, i.e. A[i] ≤ A[i+1]
then we can use BINARY SEARCH

\[ BS(k, m) = \begin{cases} 
\text{RETURN (YES)} & \text{if } x = q = A\left\lfloor \frac{m-k}{2} \right\rfloor \\
BS(k, \lceil m/2 \rceil - 1) & \text{if } x < q \\
BS(\lceil m/2 \rceil + 1, m) & \text{if } x > q 
\end{cases} \]

CALL BS(1, n).
\[ T(n) = \begin{cases} \begin{array}{ll} C & n \leq 1 \\ 2T(n/2) + C & n > 1 \end{array} \end{cases} \]

By Master Theorem, \( O(\log n) \), i.e. \( O(\log n) \)

- Suppose only \( A[i], \ldots, A[m] \), where \( m \leq n \) are used, and the value of \( m \) is known.
- Add \( x \) to \( A \): \( A[m+1] = x \), i.e. \( O(1) \)

- Suppose \( A[i], \ldots, A[m] \) is sorted
  - Add \( x \) to \( A \):
    1. Find a place where \( x \) should be located.
    2. If this place is different then \( A[m+1] \) say \( A[k] \), move each \( A[i] \) to \( A[i+1] \) for all \( i \geq k \).

Complexity \( O(n) \), see the case if \( k = 1 \).
• Sometimes arrays are better.
• Sometimes lists are better.
• In reality all lists are implemented as arrays!
IMPLEMENTING THE STABLE MATCHING ALGORITHM,

- WE WANT TO IMPLEMENT IT IN $O(n^2)$ TIME!

- PREFERENCES: BOTH MEN AND WOMEN ARE REPRESENTED BY NUMBERS $1, 2, ..., n$,
  ManPref[$m, i$] denote the $i$th woman on $m$'s preference list,
  WomanPref[$w, i$] denote the $i$th man on $w$'s preference list.

- WE NEED THE FOLLOWING FOUR THINGS TO BE EXECUTED IN $O(1)$

1. TO IDENTIFY A FREE MAN
2. FOR A MAN $m$ TO IDENTIFY THE HIGHEST-RANKED WOMAN TO WHOM HE HAS NOT YET PROPOSED
3. FOR A WOMAN $w$, WE NEED TO DECIDE IF $w$ IS CURRENTLY ENGAGED, AND IF SHE IS, WE NEED TO IDENTIFY HER CURRENT PARTNER
4. FOR A WOMAN $w$ AND TWO MEN $m, m'$, WE NEED TO BE ABLE TO DECIDE, WHICH ONE IS PREFERRED BY $w$. 
(1) IDENTIFICATION OF A FREE MAN

\[ \text{FreeM} = [m_1] \rightarrow [m_2] \rightarrow \ldots \rightarrow [m_k] \]

\[ m = \text{FIRST} (\text{FreeM}) \]

\[ \text{INSERT}(m') \text{ if } m' \text{ becomes free} \]

Clearly \( O(1) \)

- POPULAR SOLUTION TO SIMILAR PROBLEMS, AS FREE MEMORY!

(2) HIGHEST-RANKED WOMAN TO WHOM \( m \) HAS NOT YET PROPOSED.

EXTRA ARRAY \( \text{Next} \)

\[ \text{Next}[m] = \text{the position of the next woman he will propose to on his list} \]

Initially \( \text{Next}[m] = 1 \).

\( m \) always proposes to \( w = \text{MaxPrev}[m, \text{Next}[m]] \) and then \( \text{Next}[m] = \text{Next}[m] + 1 \), regardless of whether or not \( w \) accepts the proposal.

Hence \( O(1) + O(1) = O(1) \)
(3) IS W CURRENTLY ENGAGED AND IF YES TO WHOM?

EXTRA ARRAY Current

Current[w] = m if w is engaged to m
Initially Current[w] = 0 for all w.

If w becomes engaged to m, Current[w] = m.
Clearly O(1)

(4) For m and m', which one is preferred by w.

Not so obvious!
Checking WomanPref[w, m] does not work so it require O(n) time, while we need O(1)!

Extra array Ranking[w, m].

Ranking[w, m] = i, if m is on the ith position on w list, i.e., if WomanPref[w, i] = m.

Ranking can be create from WomanPref in O(n^2) time.

Ranking is created once at the beginning.
Then w → m, m' if and only if
Ranking[w, m] > Ranking[w, m']

Hence O(1) again.
GS ALGORITHM CAN BE IMPLEMENTED IN $O(n^2)$ TIME

- IMPLEMENTATION CONSISTS OF TWO PARTS
  
  1) CREATING APPROPRIATE DATA STRUCTURES
     
     ManPrev $[\cdot, \cdot]$  $O(n^2)$
     WomanPrev $[\cdot, \cdot]$  $O(n^2)$
     FreeMan = $\begin{array}{c}
     m_1 \\
     \rightarrow \\
     m_2 \\
     \rightarrow \\
     \vdots \\
     m_n \end{array}$  $O(n)$
     Next $[\cdot]$  $O(n)$
     Current $[\cdot]$  $O(n)$
     Ranking $[\cdot, \cdot]$  $O(n^2)$
     Total  $O(n^2)$

  2) IMPLEMENTING THE LOOP WHILE WHICH IS $O(n^2)$ ($O(1) + O(1) + O(1) + O(1)) = O(n^2)$
     Total  $O(n^2)$

$O(n^2) + O(n^2) = O(n^2)$

FINAL TIME ESTIMATION
Most of the algorithms' implementations can be divided into:

1) Preparation of data structures
2) Running the algorithm.

Note that space complexity is for GS algorithm $O(n^2)$ as well. More precisely it is smaller than

$3n^2 + 3n + k$

where $k$ is the number of single variables used in a program.
Data Structures

- **ABSTRACT DATA TYPES:**
  
  Set of objects + a mathematical model with a collection of operations defined on the model.

- **Dynamic set** (dictionary)
  
  One set $S$, elements of $S$ might have an attribute "key", i.e. $x.key$, $S$ might be totally ordered.

querys: • modifiers: •

- **SEARCH($S, k$):** returns a pointer $p$ such that $p \rightarrow x$ and $x.key = k$, or NIL if such $x$ does not belong to $S$.

- **INSERT($S, x$):** adds $x$ to $S$, $S \leftarrow S \cup \{x\}$

- **DELETE($S, x$):** $S \leftarrow S \setminus \{x\}$

- **MIN($S$):** $S$ must be totally ordered (with comparison)

- **MAX($S$):**

- **SUCCESSOR($S, x$):**

- **PREDECESSOR($S, x$):**
• **ADT** STACK

• Last-In First-Out policy (LIFO)

• Operations: PUSH POP EMPTY TOP

Array implementation:

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
15 & 6 & 2 & \_ & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
15 & 6 & 2 & 9 & 17 & 17 & 17 \\
\end{array}
\]

- \( \text{top}[s] = 4 \)
- \( \text{PUSH}(s, 17) \)
- \( \text{top}[s] = 5 \)
- \( \text{PUSH}(s, 3) \)
- \( \text{top}[s] = 6 \)
- \( \text{POP} \)
- \( \text{top}[s] = 5 \)

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
15 & 6 & 2 & 9 & 17 & 3 & 3 \\
\end{array}
\]

**EMPTY(s)**

\[ \text{if} \ \text{top} = 0 \ \text{then return (TRUE)} \ \text{else return (FALSE)} \]

**PUSH(s, x)**

\[ \text{top} \leftarrow \text{top} + 1; \]
\[ s[\text{top}] \leftarrow x \]

**POP(s)**

\[ \text{if EMPTY(s) then error ("underflow") else begin top} \leftarrow \text{top} + 1; \]
\[ \text{return } s[\text{top}] \text{ end} \]
- **ADT QUEUE**

- **First-In First-Out policy (FIFO)**

- **Operations:** ENQUEUE, DEQUEUE

  **FIRST, EMPTY**

**Array implementation**

Queue has 2 attributes: head and tail

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

**ENQUEUE (17)**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

**ENQUEUE (3)**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

**ENQUEUE (5)**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

**DEQUEUE**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>
let $Q$ be an array $Q[1...n]$.

$$i \oplus 1 = \begin{cases} i+1 & i < n \\ 1 & i = n \end{cases}$$

- tail = head $\iff$ $Q$ is empty
- head = tail $\oplus$ 1 $\iff$ $Q$ is full
- initially: head = tail = 1

- the elements in the queue are in locations:
  head, head+1, ..., tail-1

- an array may hold at most $n-1$ elements.

**ENQUEUE** ($x$)

if head = tail $\oplus$ 1 then error ("overflow")

else begin

  $Q[tail] \gets x$;

  tail $\gets$ tail $\oplus$ 1

end

**DEQUEUE**

if tail = head then error ("underflow")

else begin

  $x \gets Q[head]$;

  head $\gets$ head $\oplus$ 1;

  return ($x$)

end
EMPTY
if tail = head then return (TRUE)
else return (FALSE)

FIRST
if EMPTY then error ("underflow")
else return (Q[head])

* CIRCULAR BUFFER:
ADT LISTS (REPRESENT DYNAMIC SET)

1. singly linked (unsorted)

```
head -> 9 -> 16 -> 4 -> 1
```

2. double linked (unsorted)

```
head -> 15 <-> 16 <-> 4 <-> 1
```

3. singly sorted

```
1 -> 4 -> 9 -> 16
```

4. double sorted

```
11 <-> 4 <-> 9 <-> 16
```
- **CIRCULAR**  (UNSORTED)

![Circular linked list diagram]

- **OPERATIONS DIFFERS SLIGHTLY FOR DIFFERENT TYPES OF LISTS.**

- **Case of unsorted double linked**

```plaintext
SEARCH (L, k) ← finds first x with x.key = k
x ← head;
while x ≠ NIL and x.key ≠ k
    do x ← next(x)
return x
```

**Complexity** $\mathcal{O}(n)$
\[
\text{INSERT}(x) \leftarrow \text{adds } x \text{ at the beginning}
\]
\[
\text{next}(x) \leftarrow \text{head};
\]
\[
\text{if } \text{head} \neq \text{NIL} \text{ then } \text{prev}(\text{head}) \leftarrow x;
\]
\[
\text{head} \leftarrow x;
\]
\[
\text{prev} \leftarrow \text{NIL};
\]

**Complexity**: \(O(1)\)

In the notation above \( x = \begin{array}{c}
\text{prev key next}
\end{array} \)

Notation differs:
\[
\begin{align*}
\text{x.prev} & \equiv \text{prev}(x) \equiv \text{prev}[x] \\
\text{x.key} & \equiv \text{key}(x) \equiv \text{key}[x] \\
\text{x.next} & \equiv \text{next}(x) \equiv \text{next}[x]
\end{align*}
\]

\[
\text{DELETE}(x) \leftarrow \text{assumes a pointer to } x \text{ is known!!}
\]
\[
\text{if } x.\text{prev} \neq \text{NIL}
\]
\[
\quad \text{then } (x.\text{prev}).\text{next} \leftarrow x.\text{next}
\]
\[
\text{else head} \leftarrow x.\text{next}; \quad \text{complexity } O(1)
\]
\[
\text{if } x.\text{next} \neq \text{NIL}
\]
\[
\quad \text{then } (x.\text{next}).\text{prev} \leftarrow x.\text{prev};
\]

\[
\text{DELETE KEY}(k)
\]
\[
\text{x} \leftarrow \text{SEARCH}(k);
\]
\[
\text{DELETE}(x) \quad \text{complexity } O(n)
\]
SENTINELS

They make operations easier, and can hold some aggregate information about a list.

Adding sentinels makes lists circular.

Average value of a key
Implementing pointers and objects.

- Multiple-array representation

```
+---+   +---+   +---+
| 9 |   | 16 |   | 4 |
+---+   +---+   +---+
   |     |     |     |
prev↑   key↑   next↑
   |     |     |     |
   ↓     ↓     ↓
   prev   next   x
   |       |       |
   v       v       v
   |       |       |
   v       v       v
   |       |       |
   v       v       v
   |       |       |
```

```
1  2  3  4  5  6  7  8  9 10 11 12 13
| 4 | 1 | 16 | 9 |
5  2  7  1
+---+   +---+   +---+   +---+
```

an object \( x \) with a pointed (cursor) \( p = 11 \) and \( p \rightarrow x \)
• Single array implementation
ALLOCATING AND FREEING OBJECTS

- Free list: a stack \( \text{free} \Leftrightarrow \text{top} \)

\begin{equation}
\text{ALLOCATE-OBJECT}(\ )
\end{equation}

\[
\text{if } \text{free} = \text{NIL} \text{ then enov("out of space")}
\]
\[
\text{else begin}
\]
\[
\text{x } \leftarrow \text{free;}
\]
\[
\text{free } \leftarrow \text{next(x)}
\]
\[
\text{end}
\]

\begin{equation}
\text{FREE-OBJECT}(x)
\end{equation}

\[
\text{next}(x) \leftarrow \text{free}
\]
\[
\text{free } \leftarrow x
\]
**Allocate()**

- **L = 9, 4, 16, 1**
- **Free = 4, 8, 6, 1**

**Allocate()** returns 4

- **L = 7, 2, 5, 3**
- **Free = 8, 6, 1**
- current pointer = 4

**Insert(L, 25, 4)**

- **Insert(L, 25, Allocate())**

- It inserts 25 at position 4
- **Insert(L, key, p)** inserts key at position pointed to by p, and p becomes the first element of the list!

- **L = 4, 7, 2, 5, 3**
  - 25, 9, 4, 16, 1
- **Free = 8, 6, 1**
DELETE(5)
DELETE KEY(16)

\[ L = 4, 7, 2, 3, 25, 9, 4, 1 \]

\[ \text{free} = 5, 8, 6, 1 \]

- In reality, we often allocate more than one object! `malloc` in C.
**MERGE SORT**

- **Operation** **merge**:

  **input**: two sorted lists
  **output**: one collective sorted list

- **Example**:

  1 5 7 8 14 18 21 21
  2 7 14 17 20 22
  1
  
  1 5 7 8 14 18 21 21
  2 7 14 17 20 22
  1 2
  
  1 5 7 8 14 18 21 21
  2 7 14 17 20 22
  1 2 5

  and so on
We always move cursors to the right.

Complexity of merge is \( O(n) \)

- **Merge Sort** (non-recursive)

\[
\begin{align*}
15 & 3 & 5 & 16 & 28 & 14 & 2 & 1 & 18 & 9 & 7 & 5 \\
3 & 15 & 5 & 16 & 14 & 28 & 12 & 9 & 18 & 5 & 7 \\
\text{merge} & & & & & & & & & & & \\
3 & 5 & 15 & 16 & 1 & 2 & 14 & 28 & 5 & 7 & 9 & 18 \\
\text{merge} & & & & & & & & & & & \\
1 & 2 & 3 & 5 & 14 & 15 & 16 & 57 & 9 & 18
\end{align*}
\]
Complexity $O(\log_2 n) \cdot O(n) = O(n \log n)$
Recursive Merge Sort (2-way Merge Sort)

Algorithm MSort (A [1..n])

if \( n > 1 \) then begin

\[ m \leftarrow \lceil n/2 \rceil; \]
MSort (A [1..m]);
MSort (A [m+1..n]);
Merge (A [1..m], A [m+1..n], B [1..n]);
A [1..n] \leftarrow B [1..n];
end

\[
T(n) = 2T(n/2) + cn
\]

\[ \uparrow \quad \text{two sorts} \quad \uparrow \quad \text{cost of merge} \]

\[ = 4T(n/4) + cn + \frac{2cn}{2} = 4T(n/4) + 2cn \]

\[ = 8T(n/8) + 3cn = \cdots \]

\[ = 2^\log n \cdot O(1) + (\log n)cn = O(n\log n) \]

\[ \uparrow \quad \text{T(1)} \]

Similarly, \( T(n) = \Omega(n\log n) \)
Martian Theorem

\( a = 2, \ b = 2, \ \log_2 2 = 1, \quad f(n) = n \)

so \( f(n) = \Theta(n') \), i.e. case 2.

\[ T(n) = \Omega(n \log \log n) = \Omega(n \log n) \]