Complexity

**Objective:** Check the feasibility of using an algorithm for solving a particular class of problem
Complexity

**Objective:** Check the feasibility of using an algorithm for solving a particular class of problem

**Types:** Worst case - no input does any worse than this
          Average case - average performance
Complexity

**Objective**: Check the feasibility of using an algorithm for solving a particular class of problem

**Types**: Worst case - no input does any worse than this

Average case - average performance

**Input Size**: Say number of cities or cables, etc. \((n,m)\)
Complexity

**Objective**: Check the feasibility of using an algorithm for solving a particular class of problem

**Types**: Worst case - no input does any worse than this

Average case - average performance

**Input Size**: Say number of cities or cables, etc. \((n,m)\)

**Measure**: Some key operation determining behavior, as a function of input size
Complexity

Objective: Check the feasibility of using an algorithm for solving a particular class of problem

Types: Worst case - no input does any worse than this
       Average case - average performance

Input Size: Say number of cities or cables, etc. \((n,m)\)

Measure: Some key operation determining behavior, as a function of input size

Limit: Find a bound on the number of operations, in the limit, as \(n \rightarrow \infty\)
Complexity

**Example**: Given an array of integers in no particular order, find the maximum integer in the array.
Complexity

Example: Given an array of \texttt{ints} in no particular order, find the maximum \texttt{int} in the array.

Algorithm: examine by increasing index, save the largest seen so far
Complexity

Example: Given an array of ints in no particular order, find the maximum int in the array.

Algorithm: examine by increasing index, save the largest seen so far

```c
int findmax(int *elem, int n) {
    int index = 0;
    if (n < 1) return -1;
    int maxnum = elem[0];
    for (int i=0 ; i < n ; i++) {
        if (elem[i] > maxnum) {
            maxnum = elem[i];
            index = i;
        }
    }
    return index;
}
```
**Complexity**

**Operation:** Dominating operation is >

**Input Size:** Number of elements in array

**Complexity:**

```c
int findmax(int *elem, int n) {
    int index = 0;
    if (n < 1) return -1;
    int maxnum = elem[0];
    for (int i=0 ; i < n ; i++) {
        if (elem[i] > maxnum) {
            maxnum = elem[i];
            index = i;
        }
    }
    return index;
}
```
**Complexity**

**Operation:** Dominating operation is >

**Input Size:** Number of elements in array

**Complexity:** $O(n)$ – both worst, average case

```c
int findmax(int *elem, int n) {
    int index = 0;
    if (n < 1) return -1;
    int maxnum = elem[0];
    for (int i=0 ; i < n ; i++) {
        if (elem[i] > maxnum) {
            maxnum = elem[i];
            index = i;
        }
    }
    return index;
}
```
Complexity

Example: Given an array of ints in no particular order, and an int \( x \), find the first matching \( x \).

Algorithm: examine by increasing index, return array index when a match is found

```
int findint(int *elem, int n, int x) {
    if (n < 1) return -1;
    for (int i=0 ; i < n ; i++) {
        if (elem[i] == x) return i;
    }
    return -1;
}
```
Complexity

Operation: Dominating operation is >

Input Size: Number of elements in array

Complexity: $O(n)$ – both worst, average case

```c
int findint(int *elem, int n, int x) {
    if (n < 1) return -1;
    for (int i=0 ; i < n ; i++) {
        if (elem[i] == x) return i;
    }
    return -1;
}
```
Complexity

Example: Given an array of ints in increasing order, find an int matching a given $x$.

Algorithm: binary search

```c
int binsearch(int *elem, int f, int l, int x) {
    while (f <= l) {
        int index = (f+l)/2;
        if (x == elem[index]) return index;
        if (x < elem[index]) l = (f+l)/2-1;
        else f = (f+l)/2+1;
    }
    return -1;
}
```
Complexity

**Operation**: Dominating operations are > and ==

**Input Size**: Number of elements in array

**Complexity**: $O(lg(n))$ – How to get this?
Complexity – worst case

Before starting, max number compares to do is $n$
Complexity – worst case

Before starting, max number compares to do is $n$
After 1 iteration, max number compares left = $n/2$
Complexity – worst case

Before starting, max number compares to do is $n$

After 1 iteration, max number compares left $= n/2$

After $2^{nd}$ iteration, max number compares left $= n/4$
Complexity – worst case

Before starting, max number compares to do is $n$

After 1 iteration, max number compares left = $n/2$

After 2$^{nd}$ iteration, max number compares left = $n/4$

After 3$^{rd}$ iteration, max number compares left = $n/8$

...
Complexity – worst case

Before starting, max number compares to do is $n$

After 1 iteration, max number compares left = $n/2$

After 2$^{nd}$ iteration, max number compares left = $n/4$

After 3$^{rd}$ iteration, max number compares left = $n/8$

...$

After i iterations, max number compares left = $n/2^i$

...
Complexity – worst case

Before starting, max number compares to do is $n$
After 1 iteration, max number compares left = $n/2$
After 2$^{nd}$ iteration, max number compares left = $n/4$
After 3$^{rd}$ iteration, max number compares left = $n/8$
...
After i iterations, max number compares left = $n/2^i$
...
After how many iterations is there when one compare left?
Complexity – worst case

Before starting, max number compares to do is $n$
After 1 iteration, max number compares left $= n/2$
After 2\textsuperscript{nd} iteration, max number compares left $= n/4$
After 3\textsuperscript{rd} iteration, max number compares left $= n/8$
...
After $i$ iterations, max number compares left $= n/2^i$
...
After how many iterations is there when one compare left?

Answer: when $1 = n/2^i$, or when $0 = \lg(n) - i$
Complexity – average case

Let $X$ be a random integer variable,

Let $Pr(X=i)$ denote the probability that $X$ has the value $i$.

Then $E\{X\} = Pr(X=1) + Pr(X=2) \times 2 + Pr(X=3) \times 3 + ...$
Complexity – average case

Assume one we look for is in the array
Complexity – average case

Assume one we look for is in the array

Probability(land on it, 1\textsuperscript{st} iteration) = 1/n
Complexity – average case

Assume one we look for is in the array

Probability(land on it, 1\textsuperscript{st} iteration) = \frac{1}{n}

Probability(land on it, 2\textsuperscript{nd} iteration) = \frac{1}{(n/2)}
Complexity – average case

Assume one we look for is in the array

Probability(land on it, 1\textsuperscript{st} iteration) = 1/n
Probability(land on it, 2\textsuperscript{nd} iteration) = 1/(n/2)
Probability(land on it, 3\textsuperscript{rd} iteration) = 1/(n/4)
Complexity – average case

Assume one we look for is in the array
Probability(land on it, 1\textsuperscript{st} iteration) = 1/n
Probability(land on it, 2\textsuperscript{nd} iteration) = 1/(n/2)
Probability(land on it, 3\textsuperscript{rd} iteration) = 1/(n/4)

Hence average number compares =

\((1/n)*1 + (2/n)*2 + (4/n)*3 + (8/n)*4 + \ldots + (2^{\log(n)-1}/n)*\log(n)\)
Complexity – average case

\[
\frac{1}{n} + \frac{2}{n} + \frac{4}{n} + \frac{8}{n} + \ldots + \frac{(n/2)}{n} + \\
\frac{2}{n} + \frac{4}{n} + \frac{8}{n} + \ldots + \frac{(n/2)}{n} + \\
\frac{4}{n} + \frac{8}{n} + \ldots + \frac{(n/2)}{n} + \\
\ldots \\
+ \frac{(n/2)}{n}
\]
Complexity – average case

\[
\frac{1}{n} + \frac{2}{n} + \frac{4}{n} + \frac{8}{n} + \ldots + \frac{(n/2)}{n} + \frac{2}{n} + \frac{4}{n} + \frac{8}{n} + \ldots + \frac{(n/2)}{n} + \frac{4}{n} + \frac{8}{n} + \ldots + \frac{(n/2)}{n} + \ldots + \frac{(n/2)}{n}
\]

But

\[
1 + 2 + 4 + 8 + \ldots + n = 2n - 1
\]

\[
1 + 2 + 4 + 8 + \ldots + \left(\frac{n}{2}\right) = n - \frac{1}{2}
\]

<table>
<thead>
<tr>
<th>(n)</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
</tr>
</tbody>
</table>
Complexity – average case

\[ \frac{1}{n} + \frac{2}{n} + \frac{4}{n} + \frac{8}{n} + ... + \frac{(n/2)}{n} + \]
\[ \frac{2}{n} + \frac{4}{n} + \frac{8}{n} + ... + \frac{(n/2)}{n} + \]
\[ \frac{4}{n} + \frac{8}{n} + ... + \frac{(n/2)}{n} + \]
\[
\]
\[ ... \]
\[ + \frac{(n/2)}{n} \]

But

\[
\begin{array}{c|c}
 n & \text{sum} \\
1 & 1 \\
2 & 3 \\
4 & 7 \\
8 & 15 \\
\end{array}
\]

\[1 + 2 + 4 + 8 + ... + \frac{(n/2)}{n} = n - \frac{1}{2} \]

\[2 + 4 + 8 + ... + \frac{(n/2)}{n} = n - 1 \]

\[4 + 8 + ... + \frac{(n/2)}{n} = n - 2 \]

\[8 + ... + \frac{(n/2)}{n} = n - 4 \]
Complexity – average case

\[
\frac{1}{n} + \frac{2}{n} + \frac{4}{n} + \frac{8}{n} + \ldots + \frac{(n/2)}{n} + \\
\frac{2}{n} + \frac{4}{n} + \frac{8}{n} + \ldots + \frac{(n/2)}{n} + \\
\frac{4}{n} + \frac{8}{n} + \ldots + \frac{(n/2)}{n} + \\
\vdots + \frac{(n/2)}{n}
\]

So

\[
(1 - \frac{1}{2n}) + (1 - \frac{1}{n}) + (1 - \frac{2}{n}) + (1 - \frac{4}{n}) + \ldots + 1 \\
= \lg(n) - 1 = O(\lg(n))
\]
Complexity

Example: Insertion sort

```c
void insertSort(int *array_beg, int *array_end) {
    int i;
    if (array_beg == array_end) return;
    insertSort(array_beg+1, array_end);
    insert(array_beg, array_end);
}

void insert(int *first, int *last) {
    int *ptr, a = *first;
    for (ptr = first+1 ; ptr != last ; ptr++) {
        if (*ptr > a) {
            *(ptr-1) = a;
            return;
        }
        *(ptr-1) = *ptr;
    }
    if (*ptr > a) *(ptr-1) = a;
    else {
        *(ptr-1) = *ptr;
        *ptr = a;
    }
}
```
Complexity

Complexity of insert: List of $i$ elements: $O(i)$

Complexity of insertion sort:

Let $T(n)$ be the complexity on a list of $n$ elements
Complexity

Complexity of insert: List of $i$ elements: $O(i)$

Complexity of insertion sort:

Let $T(n)$ be the complexity on a list of $n$ elements

$$T(n) = T(n-1) + n; \quad T(1)=1$$
Complexity

Complexity of insert: List of \( i \) elements: \( O(i) \)

Complexity of insertion sort:

Let \( T(n) \) be the complexity on a list of \( n \) elements

\[
T(n) = T(n-1) + n; \quad T(1) = 1
\]

\[
T(1) = 1
\]

\[
T(2) = T(1) + 2
\]

\[
T(3) = T(2) + 3
\]

\[
T(4) = T(3) + 4
\]

... 

\[
T(n) = T(n-1) + n
\]
Complexity

**Complexity of insert**: List of $i$ elements: $O(i)$

**Complexity of insertion sort**:

Let $T(n)$ be the complexity on a list of $n$ elements.

$$T(n) = T(n-1) + n; \ T(1)=1$$

- $T(1) = 1$
- $T(2) = T(1) + 2$
- $T(3) = T(2) + 3$
- $T(4) = T(3) + 4$

... ...

- $T(n) = T(n-1) + n$

$$T(n) = 1+2+3+4+\ldots+n = n(n-1)/2$$
Complexity - average

Complexity of insert: List of $i$ elements: $i/2$

Complexity of insertion sort:

Let $T(n)$ be the average complexity on a list of $n$ elements

$$T(n) = T(n-1) + n/2; \quad T(1) = 1$$

$$T(1) = 1$$
$$T(2) = T(1) + 1$$
$$T(3) = T(2) + 1.5$$
$$T(4) = T(3) + 2$$

...$

$$T(n) = T(n-1) + n/2$$

$$T(n) = (1+2+3+4+...+n)/2 = n(n-1)/4$$
Complexity

Example: Mergesort

```c
void mergesort (int *A, int f, int l) {
    if (f >= l) return;
    mergesort(A, f, (f+l)/2);
    mergesort(A, (f+l)/2+1, l);
    merge(A, f, (f+l)/2, l);
}
```

```c
void merge(int *A, int f, int mid, int l) {
    int temp[n], f1=f, l1=mid, f2=mid+1, l2 = l;
    int i;
    for (i=f1; f1 <= l1 && f2 <= l2; i++) {
        else temp[i] = A[f2++];
    }
    for ( ; f1 <= l1 ; f1++, i++) temp[i] = A[f1];
    for ( ; f2 <= l2 ; f2++, i++) temp[i] = A[f2];
    for (i=f ; i <= l ; i++) A[i] = temp[i];
}
```
Complexity

Complexity of merge: If both lists together have \( i \) elements, the complexity of merge is \( O(i) \)

Complexity of mergesort:
Let \( T(n) \) be the complexity on a list of \( n \) elements
Complexity

**Complexity of merge**: If both lists together have $i$ elements, the complexity of merge is $O(i)$

**Complexity of mergesort**: Let $T(n)$ be the complexity on a list of $n$ elements

$$T(n) = 2T(n/2) + n \; ; \; T(1) = 1$$
Complexity

Complexity of merge: If both lists together have \( i \) elements, the complexity of merge is \( O(i) \)

Complexity of mergesort:
Let \( T(n) \) be the complexity on a list of \( n \) elements

\[
T(n) = 2T(n/2) + n ; T(1) = 1
\]

\[
T(1) = 1 \\
T(2) = 2T(1) + 2 \\
T(4) = 2T(2) + 4 \\
T(8) = 2T(4) + 8 \\
\]

\[
\ldots
\]

\[
T(2^{\lg(n)}) = 2T(2^{\lg(n)-1}) + 2^{\lg(n)}
\]
Complexity

\[ T(1) = 1 \]
\[ (T(2) = 2T(1) + 2)/2 \]
\[ (T(4) = 2T(2) + 4)/4 \]
\[ (T(8) = 2T(4) + 8)/8 \]

\[ \ldots \]
\[ (T(2^{\lg(n)}) = 2T(2^{\lg(n)-1}) + 2^{\lg(n)})/2^{\lg(n)} \]
Complexity

\[ T(1) = 1 \]
\[ (T(2) = 2T(1) + 2)/2 \]
\[ (T(4) = 2T(2) + 4)/4 \]
\[ (T(8) = 2T(4) + 8)/8 \]

... 

\[ (T(2^{\log(n)}) = 2T(2^{\log(n)-1}) + 2^{\log(n)})/2^{\log(n)} \]

so

\[ T(2^{\log(n)})/2^{\log(n)} = 1 + 1 \ldots + 1 \]

\[ T(n) = \log(n) \cdot n \]
Average Complexity

Let $T(n)$ be the average time to sort $n$ numbers.
Let $X_n$ be the number of compares in a merge of two $n/2$ size lists.
Average Complexity

Let $T(n)$ be the average time to sort $n$ numbers.

Let $X_n$ be the number of compares in a merge of two $n/2$ size lists.

$X_n$ could be as low as $n/2$ and as high as $n$.

<table>
<thead>
<tr>
<th>1 2 3 4</th>
<th>6 7 8 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 6 8</td>
<td>3 4 7 9</td>
</tr>
</tbody>
</table>
Average Complexity

Let $T(n)$ be the average time to sort $n$ numbers. Let $X_n$ be the number of compares in a merge of two $n/2$ size lists. $X_n$ could be as low as $n/2$ and as high as $n$.

Successive dots represent successively larger numbers in both lists. Color of dot is the list the number is in. First color change from the right determines number of compares: e.g. $X_{14} = 13-2$ above.
Average Complexity

Let $T(n)$ be the average time to sort $n$ numbers.

Let $X_n$ be the number of compares in a merge of two $n/2$ size lists.

$X_n$ could be as low as $n/2$ and as high as $n$.

Successive dots represent successively larger numbers in both lists.
Color of dot is the list the number is in. First color change from the right determines number of compares: e.g. $X_{14} = 13-2$ above.

Hence $Pr(X_n = n-x) = \binom{n-x-1}{n/2-1}/\binom{n}{n/2} = \frac{(n-x-1)!(n/2)!(n/2)!}{n!(n/2-1)!(n/2-x)!} \approx 2^{-x}$

$E\{X_n\} = \sum_{1\leq x \leq n/2} (n-x)2^{-x} = n -1/2 -2/4 -3/8 -4/16 -... = n-2$
Average Complexity

\[ T(n) = 2T(n/2) + E\{X_n\} ; \quad T(1) = 1 ; \quad T(2) = 1 \]
Average Complexity

\[ T(n) = 2T(n/2) + E\{X_n\} \ ; \ T(1) = 1 \ ; \ T(2) = 1 \]

\[ T(n) = 2T(n/2) + n - 2 \]
Average Complexity

\[ T(n) = 2T(n/2) + E\{X_n\} ; T(1) = 1 ; T(2) = 1 \]

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

\[ T(1) = 1 \]
\[ T(2) = 1 \]
\[ T(4) = 2T(2) + 2 \]
\[ T(8) = 2T(4) + 6 \]

\[ \ldots \]
\[ T(2^{\lg(n)}) = 2T(2^{\lg(n)-1}) + 2^{\lg(n)} - 2 \]
Average Complexity

\[ T(n) = 2T(n/2) + E\{X_n\} \ ; \ T(1) = 1 \ ; \ T(2) = 1 \]

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

\[ T(1) = 1 \]
\[ T(2) = 1 \]
\[ T(4) = 2T(2) + 2 \]
\[ T(8) = 2T(4) + 6 \]

... 

\[ T(2^{\lg(n)}) = 2T(2^{\lg(n)-1}) + 2^{\lg(n)} - 2 \]

\[ T(2^{\lg(n)})/2^{\lg(n)-1} = 2+(2-1)+(2-.5)+(2-.25)+(2-.125)+...+2-(2/n) \]
\[ = 2*\lg(n) - 2 + 2/n \]
Average Complexity

\[ T(n) = 2T(n/2) + E\{X_n\} \ ; \ T(1) = 1 \ ; \ T(2) = 1 \]

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

\[ T(1) = 1 \]
\[ T(2) = 1 \]
\[ T(4) = 2T(2) + 2 \]
\[ T(8) = 2T(4) + 6 \]

... \]

\[ T(2^{\lg(n)}) = 2T(2^{\lg(n)-1}) + 2^{\lg(n)} - 2 \]

\[ T(2^{\lg(n)})/2^{\lg(n)-1} = 2+(2-1)+(2-.5)+(2-.25)+(2-.125)+...+2-(2/n) \]
\[ = 2*\lg(n) - 2 + 2/n \]

\[ T(n) = n*(\lg(n)-1) + 1 \]
Complexity

Quicksort:

```c
void partition(int *A, int f, int l, int *piv_idx){
    int pivot = A[f], last = f, first;
    for (first=f+1 ; first <= l ; first++) {
    }
    swap(&A[f], &A[last]);
    *piv_idx = last;
}

void Quicksort(int *A, int f, int l) {
    int pivot_index;
    if (f >= l) return;
    if (f >= l) return;
    partition(A, f, l, &pivot_index);
    Quicksort(A, f, pivot_index-1);
    Quicksort(A, pivot_index+1, l);
}```
Complexity

Quicksort worst case:

Let $T(n)$ denote the worst case time for Quicksort

\[ T(n) = 1 + T(n-1) + n; \quad T(1) = 1 \]

\[

t(1) = 1 + 1 \\
T(2) = 1 + T(1) + 2 \\
T(3) = 1 + T(2) + 3 \\
T(4) = 1 + T(3) + 4 \\
\ldots \\
T(n) = 1 + T(n-1) + n
\]

Hence

\[
1 + 1 + \ldots + 1 + 1 + 2 + 3 + 4 + \ldots + n = n + n(n-1)/2
\]
Average Complexity

General case:
Let $T(n)$ denote the average case time for an algorithm
Let $E(0), E(1), \ldots, E(n-1)$ be a sequence of possible first outcomes in running the algorithm and let $T(E(0)), T(E(1)), \ldots, T(E(n-1))$ be the average times given the indicated event.

Then $T(n) = T(E(0)) \cdot Pr(E(0)) + T(E(1)) \cdot Pr(E(1)) + \ldots T(E(n-1)) \cdot Pr(E(n-1))$
Average Complexity

Example:

<table>
<thead>
<tr>
<th>Group</th>
<th>Scores</th>
<th>Avg</th>
<th>Prob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nerds</td>
<td>78  96 83 86</td>
<td>85.75</td>
<td>.2666</td>
</tr>
<tr>
<td>Noobs</td>
<td>14  29 73 58 71</td>
<td>49.00</td>
<td>.3333</td>
</tr>
<tr>
<td>Normals</td>
<td>56  69 88 73 49 81</td>
<td>69.33</td>
<td>.4000</td>
</tr>
</tbody>
</table>

class average = \( \frac{1004}{15} = 66.9333 \)

\[ 85.75 \times .2666 + 49 \times .3333 + 69.33 \times .4 = 66.93 \]
Complexity

Quicksort average case:

Let $T(n)$ denote the average case time for Quicksort

$$T(n) = (T(0)+T(n-1)+n)Pr(\text{pivot at position 0}) +$$
$$ (T(1)+T(n-2)+n)Pr(\text{pivot at position 1}) +$$
$$ (T(2)+T(n-3)+n)Pr(\text{pivot at position 2}) +$$

... 
$$ (T(n-1)+T(0)+n)Pr(\text{pivot at position } n-1)$$

But $Pr(\text{pivot at position } i) = 1/n$ for all $i$

So

$$T(n) = (2/n) \sum_{0 \leq i < n} T(i) + n$$
Complexity

QuickSort average case:

\[ T(n) = \frac{2}{n} \sum_{0 \leq i < n} T(i) + n \]

\[ nT(n) = 2 \sum_{0 \leq i < n} T(i) + n^2 \]

\[ nT(n) - (n-1)T(n-1) = n^2 - (n-1)^2 + 2T(n) = 2n - 1 + 2T(n) \]

\[ (n-2)T(n) = 2n + (n-1)T(n-1) - 1 \]

\[ T(n) = ((n-1)/(n-2))T(n-1) + (2n-1)/(n-2) \]

\[ T(1) = 1 \]

\[ T(2) = 1 \]

\[ T(3) = 5 + 2T(2) = 7 \]

\[ T(4) = 7/2 + (3/2)T(3) = 14 \]

\[ \ldots \]

\[ T(n) = n\log(n) \]
Complexity – Pattern Matching

Find first occurrence of string \( y \) in \( x \)

\[
x: \quad \text{abaabaabbaab} \\
y: \quad \text{aabbaaab}
\]
Complexity – Pattern Matching

Find first occurrence of string y in x

x:  abaabaabbaab
y:  aabbaab
Find first occurrence of string $y$ in $x$

$x$: abaabaabbaab
$y$: aabbaab
Find first occurrence of string y in x

\[ x: \text{abaabaabbaab} \]
\[ y: \text{aabbaab} \]
Complexity – Pattern Matching

Find first occurrence of string \( y \) in \( x \)

\[
\begin{align*}
\text{x:} & \quad \text{abaabaabbaab} \\
\text{y:} & \quad \text{aabbaab}
\end{align*}
\]
Find first occurrence of string \( y \) in \( x \)

\[
\begin{align*}
x & : \text{ abaabaabbaaab } \\
y & : \text{ aabbaab }
\end{align*}
\]
Complexity – Pattern Matching

Find first occurrence of string y in x

\[ x: \text{abaabaabbaab} \]
\[ y: \text{aabbaab} \]
Complexity – Pattern Matching

Find first occurrence of string $y$ in $x$

$x$: abaabaabbaab
$y$: aabbaab
Find first occurrence of string $y$ in $x$

$x$: abaabaabbaaab

$y$: aabbaab
Find first occurrence of string $y$ in $x$

$x$: abaabaabbaab
$y$: aabbaaab
Complexity – Pattern Matching

Find first occurrence of string y in x

x: abaabaabbaab
y: aabbaab
Complexity – Pattern Matching

Find first occurrence of string $y$ in $x$
Complexity – Amortized Analysis

Find first occurrence of string $y$ in $x$

\[ x: \quad \text{abaabaabbaab} \]
\[ y: \quad \text{aabbaab} \]
Complexity – Amortized Analysis

Find first occurrence of string $y$ in $x$

$x$: abaabaabbaab
$y$: aabbaab
Find first occurrence of string y in x

x:  abaabaabbaab
y:  aabbaab
Find first occurrence of string \( y \) in \( x \)

\[
\begin{align*}
\text{x:} & \quad \text{abaabaabbaab} \\
\text{y:} & \quad \text{aabbaaab}
\end{align*}
\]
Find first occurrence of string $y$ in $x$

\[
\begin{align*}
x & : \text{ abaabaabbaab } \\
y & : \text{ aabbaab }
\end{align*}
\]
Complexity – Amortized Analysis

Find first occurrence of string y in x

x: abaabaabbaab
y: aabbaab
Find first occurrence of string y in x

x: abaabaabbaab
y: aabbaab
Complexity – Amortized Analysis

Find first occurrence of string \( y \) in \( x \)

\( x: \) abaabaabbaab
\( y: \) aabbaab
Complexity – Amortized Analysis

Find first occurrence of string y in x

x: abaabaabbaab
y: aabbaab
Complexity – Amortized Analysis

Find first occurrence of string $y$ in $x$

$x$: abaabaabbaab
$y$: aabbaab
Complexity – Amortized Analysis

Find first occurrence of string y in x

x: abaabaabbaab
y: aabbaab
Find first occurrence of string $y$ in $x$

$x$: abaabaabbaaab
$y$: aabbaaab
Complexity – Amortized Analysis

Find first occurrence of string $y$ in $x$

$x$: abaabaabbaab
$y$: aabbaab
Find first occurrence of string $y$ in $x$

$x$: abaabaabbaab
$y$: aabbaab
Complexity – Amortized Analysis

Find first occurrence of string $y$ in $x$

$x$: abaabaabbaab
$y$: aabbaaab
Complexity – Amortized Analysis
Find first occurrence of string $y$ in $x$

$x$: abaabaabbbbaaabbaab
$y$: aabbaab
Find first occurrence of string $y$ in $x$

$x$: abaabaabbbbaabbaaabaab

$y$: aabbaab
void makeFailureFunction (char *y, int l) {
    int *f = new int[l+1];
    f[0] = f[1] = 0;
    for (int j=2 ; j < l+1 ; j++) {
        int i = f[j-1];
        while (y[j] != y[i+1] && i > 0) i = f[i];
        if (y[j] != y[i+1] && i == 0) f[j] = 0;
        else f[j] = i+1;
    }
    return f;
}

What is the complexity of this algorithm in terms of l?
void makeFailureFunction (char *y, int l) {
    int *f = new int[l+1];
    f[0] = f[1] = 0;
    for (int j=2 ; j < l+1 ; j++) {
        int i = f[j-1];
        while (y[j] != y[i+1] && i > 0) i = f[i];
        if (y[j] != y[i+1] && i == 0) f[j] = 0;
        else f[j] = i+1;
    }
    return f;
}

What is the complexity of this algorithm in terms of l?

**Answer:** $O(l)$

Cost of while is number of times $i$ is decremented in $i=f[i]$

Only way $i$ incremented: $f[j]=i+1$, $j++$, then $i=f[j-1]$

Hence $i$ decremented in while at most $l$ times