Algorithm Analysis

Computer scientists study algorithms using mathematical and empirical tools; in CS 112 we begin this study by analyzing the programs we write. The MOST important kind of behavior we want to understand is: How long does the algorithm take to finish?

We express this as a function

\[ f(N) = \text{# steps to process an input of size N} \]

But in the first place we only want to know the approximate “shape” of the function, so we count, not every step, but only “important steps” without which we can not do the algorithm.

E.g., in searching, we count the number of times we compare two keys.
Algorithm Analysis

Motivation: This is a common way of understanding the general characteristics of a function; for example in calculus, various kinds of polynomials have different shapes that determine how many roots they have, how many minima and maxima, and so on:

The function \( f(N) \) is usually some kind of polynomial or may have a \( \log_2(N) \) factor:

Example 1: Stack or Queue operations:

\[ f(N) = C \quad \text{(constant)} \]

Example 2: Searching an unordered list

\[ f(N) = A \cdot N + C \]

Example 3: Insertion sort:

\[ f(N) = A \cdot N^2 + B \cdot N + C \]

Example 4: Binary search:

\[ f(N) = A \cdot \log_2(N) + B \]
Algorithm Analysis

And we use various ways of approximating these functions, so we generally don’t need so many details:

If \( P \) is a polynomial (possibly with log terms added), then

\[
P = \sim T \quad \text{where } T \text{ is the leading term of } P \text{ (or you can write } "P \sim T"
\]

\[
P = \Theta(Q) \quad \text{where } Q \text{ is the leading term of } P \text{ without its coefficient}
\]

Examples

\[
f(N) = A*N + C = \sim (A*N) = \Theta(N)
\]

\[
f(N) = A*N^2 + B*N + C = \sim A*N^2 = \Theta(N^2)
\]

We categorize these by the degree of the leading term:

<table>
<thead>
<tr>
<th>Name</th>
<th>Formula</th>
<th>Examples:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant Time:</td>
<td>( f(N) = \sim(C) = \Theta(1) )</td>
<td>Stack push, pop</td>
</tr>
<tr>
<td>Logarithmic Time:</td>
<td>( f(N) = \sim(A*\log_2(N)) = \Theta(\log_2(N)) )</td>
<td>Binary Search in Ordered List</td>
</tr>
<tr>
<td>Linear Time:</td>
<td>( f(N) = \sim(A*N) = \Theta(N) )</td>
<td>Sequential Search in Unordered List</td>
</tr>
<tr>
<td>Linearithmic Time:</td>
<td>( f(n) = \sim(A*N^\log_2(N)) = \Theta(N^\log_2(N)) ) Mergesort (later in the lecture)</td>
<td></td>
</tr>
<tr>
<td>Quadratic Time:</td>
<td>( f(N) = \sim(A*N^2) = \Theta(N^2) )</td>
<td>Selection Sort, Insertion Sort</td>
</tr>
<tr>
<td>Cubic Time:</td>
<td>( f(N) = \sim(A*N^3) = \Theta(N^3) )</td>
<td>All Pairs Shortest Path in Graph</td>
</tr>
<tr>
<td>Exponential Time:</td>
<td>( f(N) = \Theta(2^n) )</td>
<td>Traveling Salesman Problem</td>
</tr>
</tbody>
</table>

We will study examples of algorithms in this class of all but last two....
Algorithm Analysis

To analyze an algorithm, then, you need to do the following:

Step One: Specify what “important operations” you are counting

Examples: comparing two numbers, moving a data item

Step Two: Decide whether you want to use \( \sim() \) or \( \Theta() \) or another (there are many)

In this class, we will emphasize \( \Theta() \)

Step Three: Decide whether you want to analyse the worst case, average case, or best case.

When not specified, the worst case is assumed. Best case is usually not interesting.

Step Four: Go through the algorithm, and count the number of basic operations, using the properties of \( \sim() \) and \( \Theta() \) to simplify the task......
To analyze an algorithm, you need to count how many basic operations occur in each statement, then count how many times each statement is executed. Using $\Theta$ usually simplifies this, since we can "throw out" anything that won’t contribute to the leading term without its coefficient.

Examples.... Suppose we are counting comparisons:

<table>
<thead>
<tr>
<th>Code</th>
<th>Number of comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>int [ ] A = { 2, 3, 5, 4, 6 };</td>
<td>0 (ignore any 0s)</td>
</tr>
<tr>
<td>int n = 0;</td>
<td>0</td>
</tr>
<tr>
<td>if(A[1] &lt; A[4]) ++n;</td>
<td>1 $= \Theta(1)$</td>
</tr>
<tr>
<td>if(A[0] &lt; A[3]) ++n;</td>
<td>1 $= \Theta(1)$</td>
</tr>
<tr>
<td>Total: 4 $= \Theta(1)$</td>
<td></td>
</tr>
</tbody>
</table>

Loops are essentially multiplying functions:

<table>
<thead>
<tr>
<th>Code</th>
<th>Number of comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>int [ ] A = new int[N];</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>... some code that puts values in A.....</td>
<td>$\Theta(1)$ for list of statements</td>
</tr>
<tr>
<td>Total: $\Theta(1)$</td>
<td></td>
</tr>
<tr>
<td>for(int i = 0; i &lt; N; ++i) {</td>
<td>loop repeats N times</td>
</tr>
<tr>
<td>... A list of statements (no loops)...</td>
<td>$\Theta(1)$ for list of statements</td>
</tr>
<tr>
<td>}</td>
<td>$\Theta(N)$</td>
</tr>
<tr>
<td>Total: $\Theta(N)$</td>
<td></td>
</tr>
<tr>
<td>for(int i = 0; i &lt; N; ++i) {</td>
<td>loop repeats N times</td>
</tr>
<tr>
<td>for(int j = 0; j &lt; N; ++j) {</td>
<td>loop repeats N times</td>
</tr>
<tr>
<td>.... $\Theta(1)$ for list of statements ....</td>
<td>$\Theta(1)$ for list of statements</td>
</tr>
<tr>
<td>}</td>
<td>$\Theta(N^2)$</td>
</tr>
<tr>
<td>Total: $\Theta(N^2)$</td>
<td></td>
</tr>
</tbody>
</table>
Algorithm Analysis

Graphical intuitions are often valuable, since we are abstracting away all but the basic "shape" of the way the algorithm uses the important operations. If you have \( N \) data items, the question is, how does the number of basic operations relate to \( N \)?

- **Constant Time:**
  - \( \text{if}( k < 10 ) \) { ... \( \Theta(1) \) ...
  - \( \text{for}( \text{int} \ i = 0; i < N; ++i) \) {
      - ... \( \Theta(1) \) ...
  }

- **Linear Time:**
  - \( \text{for}( \text{int} \ i = 0; i < N; ++i) \) {
      - ... \( \Theta(1) \) ...
  }

- **Quadratic Time:**
  - \( \text{for}( \text{int} \ i = 0; i < N; ++i) \) {
      - \( \text{for}( \text{int} \ j = 0; j < N; ++j) \) {
          - ... \( \Theta(1) \) ...
      }
  }

Sorting: Basic Ideas

For any set \( A \) and total ordering \( < \) on \( A \), we can define the **SORTING PROBLEM** as follows:

**Input:** A sequence \( S = ( a_0, a_1, a_2, ..., a_{n-1} ) \) of elements from \( A \)

**Output:** A permutation (rearrangement) \( ( a'_0, a'_1, a'_2, ..., a'_{n-1} ) \) of \( S \) such that

\[ a'_0 \leq a'_1 \leq a'_2 \leq ... \leq a'_{n-1} \]

**Notes:**

1. If there are no duplicates (our usual assumption) we can say:

\[ a'_0 < a'_1 < a'_2 < ... < a'_{n-1} \]

2. We will ONLY consider the problem of sorting arrays (and use arrays of integers for our illustrations).
We will study two iterative methods:

- Selection Sort
- Insertion Sort

and two recursive methods

- Merge Sort
- Quick Sort

and consider their characteristics:

- Overall approach to the problem (iterative vs. recursive);
- Use of memory (trivial except for merge sort);
- Time complexity (in terms of the number of comparisons and exchanges);
- Stability (to be defined).

Iterative methods generally have the following characteristics:

- They sort an array \( A[0..N-1] \) in place (no extra storage needed);
- They have two for loops;
  
  The outer for loop steps through the array
  
  \[
  \text{for( int } i = 0; i < N; ++i ) \{ \ldots \}
  \]

  and successively turns an unordered list into an ordered list:
Sorting: Iterative Sorting Methods

The only essential difference between our two iterative methods is how they extend the sorted region of the array:

**Selection Sort** looks for a minimal (if no duplicates, the smallest) element in the unsorted part, and moves it to the bottom of the sorted part; this is done using `exch(...)`

```java
public static void selectionSort(int[] A) {
    int N = A.length;
    for (int i = 0; i < N-1; i++) {
        int min = i;                    // assume min is A[i]
        for (int j = i+1; j < N; j++) { // look for something smaller
            if (less(A[j], A[min]))     // found one!
                min = j;
        }
        swap(A, i, min);                // swap min key and top of unsorted part
    }
}

public static boolean less( int v, int w ) {
    return ( v < w );
}

public swap( int[] A, int i, int j) {       // swap A[i] and A[j]
}
```

Sorting: Selection Sort

// Selection Sort (from Algorithms by Sedgewick and Wayne)

```java
public static void selectionSort(int[] A) {
    int N = A.length;
    for (int i = 0; i < N-1; i++) {
        int min = i;                    // assume min is A[i]
        for (int j = i+1; j < N; j++) { // look for something smaller
            if (less(A[j], A[min]))     // found one!
                min = j;
        }
        swap(A, i, min);                // swap min key and top of unsorted part
    }
}

public static boolean less( int v, int w ) {
    return ( v < w );
}

public swap( int[] A, int i, int j) {       // swap A[i] and A[j]
}
```
So, what is the Time Complexity of Selection Sort? Let us count the number of comparisons of integers, which means counting the number of times less(...) is executed (which is why we put it in a separate method).

```java
public static void selectionSort(int [] A) {
    int N = A.length;
    for (int i = 0; i < N-1; i++) {
        int min = i;    // assume min is A[i]
        for (int j = i+1; j < N; j++) { // look for something smaller
            if (less(A[j], A[min]))    // found one!
                min = j;
        }
        swap(A, i, min);
    }
}
```

```java
public static boolean less(int v, int w) {
    // could put a counter here an increment each time less is called
    return (v < w);
}
```

Complexity of Selection Sort

Let's count the number of comparisons (less):

Observe that the outer loop runs N-1 times

and less() is called once each time through the inner loop....

(N-1) times, then (N-2)....

```java
public static void selectionSort(Comparable [] a) {
    int N = a.length;
    for (int i = 0; i < N-1; i++) {
        int min = i;    // assume min is A[i]
        for (int j = i+1; j < N; j++) { // look for something smaller
            if (less(a[j], a[min]))    // found one!
                min = j;
        }
        exch(a, i, min);    // exchange min and top of
        // unsorted part
    }
}
```
Complexity of Selection Sort

We can illustrate this as follows: Let’s color the slot where the second argument to less occurs in green, and color the sorted part of the list in red:

```
7  1  1  1  1
2  2  2  2  2
1  7  7  5  5
8  8  8  8  7
5  5  5  7  8
```

\[ 4 + 3 + 2 + 1 = 10 \text{ calls to less(...)} \]

In general, for an array of size \( N \), we have

\[
(N-1) + (N-2) + \ldots + 2 + 1 = \frac{N(N-1)}{2} = \frac{N^2}{2} - \frac{N}{2} = \Theta\left(\frac{N^2}{2}\right)
\]

When \( N = 5 \), we have \( 5^2/2 - 5/2 = 25/2 - 5/2 = 20/2 = 10 \).

---

Complexity of Selection Sort

Let’s **count** the number of **comparisons** (less):

Observe that the outer loop runs \( N \) times

and \texttt{less()} is called once each time through the inner loop: \( (N-1) + \ldots + 1 = \frac{N(N-1)}{2} = \frac{N^2}{2} - \frac{N}{2} \)

```java
public static void selectionSort(Comparable [] a) {
    int N = a.length;
    for (int i = 0; i < N-1; i++) {
        int min = i;
        for (int j = i+1; j < N; j++) {
            if ( less(a[j], a[min]) )
                min = j;
        }
        exch(a, i, min);                // exchange min and top of
        // unsorted part
    }
}
```

Total: \( \frac{N^2}{2} - \frac{N}{2} = \Theta\left(\frac{N^2}{2}\right) \)
The only essential difference between our two iterative methods is how they extend the sorted region of the array:

**Insertion Sort** picks the top element of the unsorted part, and finds the place where it belongs in the sorted part, and inserts it there:

Small end

Sorted Part

Unsorted Part

Large end

K

≤ K

K

> K
Let's count the number of calls to less(...) in worst case, which in fact is a reverse sorted list....

Observe that the outer loop runs N-1 times, and less is called 1 time, then 2 times, ..... then (N-2) times, then finally (N-1) times.

```java
public static void insertionSort(int[] a) {
    int N = a.length;
    for (int i = 1; i < N; i++) {
        for (int j = i; j > 0 && less(a[j], a[j-1]); j--) {
            exch(a, j, j-1);
        }
    }
}
```

Complexity of Selection Sort

Now let's look at the diagram, coloring a slot blue if it was compared with the new key being inserted:

```
8 7 5 2 1
7 8 7 3 2
5 5 8 7 5
2 2 2 8 7
1 1 1 1 8
```

1 + 2 + 3 + 4 = 10 calls to less(...)

It is the same as for Selection Sort: \( N^2/2 - N/2 = \Theta( N^2 ) \) calls to less(....)

This is for a reverse sorted list! What about an already sorted list?
Complexity of Selection Sort

For an already sorted list, Insertion Sort does something very smart: it just checks to see that each key is not less than the one above it, and doesn’t go any further!

1 1 1 1
2 2 2 2
5 5 5 5
7 7 7 7
8 8 8 8

1 + 1 + 1 + 1 = 4 calls to less(...)

In this case, Insertion Sort checks that the list is already sorted in N-1 calls to less(...), so it is $\Theta(N) = \text{"linear time."}$

Punchline: Insertion Sort adapts to its input, and does less work than Selection Sort, except in the case of a reverse-sorted list, where they both do the same!

Algorithm Analysis

The graphical analogy helps us see why Selection sort (in all cases) is $\Theta(N^2)$:

```
for (int i = 0; i < N; i++) {
    for (int j = i+1; j < N; j++) {
        ... $\Theta(1)$ ....
    }
}
```

```
7 1 1 1 1
2 2 2 2 2
1 7 7 5 5
8 8 8 8 7
5 5 5 7 8
```

0 ... i ... N-1

$N^2 / 2$
For the **best case** of Insertion Sort, we only do one comparison per loop to check that the given item is already in the correct place, so it is $\Theta(N)$.

For the **worst case** of Insertion Sort, observe that the worst thing that can happen is each number we insert is the smallest we have seen so far:

\[
\text{insert } 1 \text{ into } 13 10 9 7 6 4 3 2
\]

So at each step of the outer loop, the new item goes all the way up to the top:
Algorithm Analysis

For the average case of Insertion Sort, observe that when we insert an arbitrary number into a ordered list, on average we go half way up:

```
insert k into 13 10 9 7 6 4 3 2
```

So at each step of the outer loop, the new item goes half way up to the top:

```
N^2 / 4
```

Iterative Sorting: Conclusions on Time Complexity

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst-case Input</th>
<th>Worst-case Time</th>
<th>Best-case Input</th>
<th>Best-case Time</th>
<th>Average-case Input</th>
<th>Average-case Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>Any!</td>
<td>Θ( N^2 )</td>
<td>Any!</td>
<td>Θ( N^2 )</td>
<td>Any!</td>
<td>Θ( N^2 )</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>Reverse Sorted List</td>
<td>Θ( N^2 )</td>
<td>Already Sorted List</td>
<td>Θ( N )</td>
<td>Random List</td>
<td>Θ( N^2 )</td>
</tr>
</tbody>
</table>

Conclusions:
- Selection Sort is inflexible and does the exact same thing in all cases;
- Insertion Sort in the worst case does no better than Selection Sort, but adapts to its input: it performs better the “more sorted” the input it; in the case of an already sorted list, it simply checks that the list is sorted.
Experimental Results

Comparing Sorting Algorithms