Lecture 17. Analysis of Algorithms

• An *algorithm* is a ‘method’ for solving a problem that is *independent* of a specific computer or programming language

• **Design**: Finding a way to solve the problem

• **Analysis**: Determining the algorithm’s cost in machine-independent terms, e.g. $\lg N$

• Need to make a program faster?
  
  Get a new machine
  
  Costs $$$ or more
  
  Makes ‘everything’ run faster
  
  But, it may — or *may not* — have much impact on a specific problem

  Get a new algorithm
  
  Costs ¢ or less
  
  Can make or break a specific problem by allowing it to be solved at all
  
  But, it may have *little or no* impact on ‘everything’
Sublist Sum Problem

- Given a list of numbers, find the contiguous sublist that has the largest sum

```
31  -41  59  26  -53  58  97  -93  -23  84
   187

31  31

31  -41  59  49

31  -41  59  26  75

31  -41  59  26  -53  22

31  -41  59  26  -53  58  80

31  -41  59  26  -53  58  97  177

31  -41  59  26  -53  58  97  -93  84

31  -41  59  26  -53  58  97  -93  -23  61

31  -41  59  26  -53  58  97  -93  -23  84  145
```

- Easy if all the numbers are nonnegative; tricky when some numbers are negative
- Sums must be positive; negative sublist sums are taken to be zero
A Simple Brute-Force Solution

• Try all possible sublists of \( n \) integers: \( x[\text{lb}..\text{ub}] \) for all \( \text{lb}, \text{ub} \) from 0 to \( n \)

```c
void sublist(int x[], int n) {
    int lb, ub, l, r, max = 0;
    for (lb = 0; lb < n; lb++)
        for (ub = lb; ub < n; ub++) {
            int i, sum = 0;
            for (i = lb; i <= ub; i++)
                sum += x[i];
            if (sum > max) {
                max = sum;
                l = lb;
                r = ub;
            }
        }
    printf("x[%d..%d] = %d\n", l, r, max);
}
```

```bash
% lcc -I/u/cs126/include sublistn3.c /u/cs126/lib/libmisc.a
% echo 31 -41 59 26 -53 58 97 -93 -23 84 | a.out
x[2..6] = 187
```
Profiling

• Program **profiles** help understand execution **frequencies**; use **lcc -b** and **bprint**

```bash
% lcc -b -I/u/cs126/include sublistn3.c /u/cs126/lib/libmisc.a
% echo 31 -41 59 26 -53 58 97 -93 -23 84 | a.out
x[2..6] = 187
% bprint
...
```

```
1  for (<1>lb = 0; <11>lb < n; <10>lb++)
2   for (<10>ub = lb; <65>ub < n; <55>ub++) {
3      int i, sum = <55>0;
        for (<55>i = lb; <275>i <= ub; <220>i++)
            <220>sum += x[i];
        if (<55>sum > max) {
            <6>max = sum;
            <6>l = lb;
            <6>r = ub;
        }
    }
 printf("x[%d..%d] = %d\n", l, r, max);
```

• For **N = 10**

<table>
<thead>
<tr>
<th>Loop</th>
<th>is executed</th>
<th>$11 \approx 10^1$ times</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>$65 \approx 10^{2/2}$</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>$275 \approx 10^{3/3}$</td>
</tr>
</tbody>
</table>

Execution time $\approx N^3$, can’t solve $N = 10,000$, since $10^{12}$ microseconds $\approx 11$ days
A Better Algorithm

• Don’t recompute the whole sum every time

\[ x[lb] + x[lb+1] + \ldots + x[ub] = (x[lb] + \ldots + x[ub-1]) + x[ub] \]

```c
void sublist(int x[], int n) {
    int lb, ub, l, r, max = 0;
    for (lb = 0; lb < n; lb++) {
        int sum = 0;
        for (ub = lb; ub < n; ub++) {
            sum += x[ub];
            if (sum > max) {
                max = sum;
                l = lb;
                r = ub;
            }
        }
    }
}
```

```c
printf("x[%d..%d] = ", l, r);
}
```

```
31 -41 59 26 -53 58 97 -93 -23 84
```

```
31 -10 49 75 22 80 177 84 61 145
```

```
31 -41 18 44 -9 49 146 53 30 114
```

```
59 85 32 90 187 94 71 155
```

```
26 -27 31 128 35 12 96
```

```
-53 5 102 9 -14 70
```

```
58 155 62 39 123
```

```
97 4 -19 65
```

```
-93 -116 -32
```

```
-23 61
```

```
84
```
Profiling the Better Algorithm

```c
for (<1>lb = 0; <11>lb < n; <10>lb++) {
    int sum = <10>0;
    for (<10>ub = lb; <65>ub < n; <55>ub++) {
        <55>sum += x[ub];
        if (<55>sum > max) {
            <6>max = sum;
            <6>l = lb;
            <6>r = ub;
        }
    }
}
printf("x[%d..%d] = %d\n", l, r, max);
```

- For $N = 10$
  
  Loop 1 is executed $11 \approx 10^1$ times
  
  Loop 2 is executed $65 \approx 10^{2/2}$ times

  Execution time $\approx N^2$, but can’t solve $N = 1,000,000$, because $10^{12}$ microseconds $\approx 11$ days

- There is a divide-and-conquer algorithm that takes $\approx N \lg N$, but there’s even a better way
The Optimal Algorithm

• Keep track of the maximum sum so far and the sum of the sublist that ends at \( x[i] \)

Suppose \( \max \) is the maximum sum in \( x[0..i-1] \); extend that solution to \( x[i] \)

\[
\begin{array}{cccccccc}
31 & -41 & 59 & 26 & -53 & 58 & 97 & -93 & -23 & 84 \\
85 & 32 & \\
31 & -41 & 59 & 26 & -53 & 58 & 97 & -93 & -23 & 84 \\
90 & \\
\end{array}
\]

```c
void sublist(int x[], int n) {
    int i, l, r, max = 0, maxi = 0;
    for (i = 0; i < n; i++) {
        if (maxi + x[i] > 0)
            maxi += x[i];
        else
            maxi = 0;
        if (maxi > max)
            max = maxi;
    }
    printf("x[%d..%d] = %d\n", l, r, max);
}
```

• Execution time \( \approx N \), because there’s just one loop; \( N = 1,000,000 \) takes \( \approx 1 \) second

• See `sublistn.c` for details of computing \( l \) and \( r \)
## Summary

- A good algorithm can be more powerful than a supercomputer

<table>
<thead>
<tr>
<th>Method</th>
<th>Time Complexity</th>
<th>Thousand</th>
<th>Million</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute Force</td>
<td>$N^3$</td>
<td>17 min</td>
<td>300 centuries!</td>
</tr>
<tr>
<td>Better</td>
<td>$N^2$</td>
<td>1 sec</td>
<td>11 days</td>
</tr>
<tr>
<td>Divide and Conquer</td>
<td>$N \log N$</td>
<td>0.01 sec</td>
<td>20 sec</td>
</tr>
<tr>
<td>Optimal</td>
<td>$N$</td>
<td>0.001 sec</td>
<td>1 sec</td>
</tr>
</tbody>
</table>

- For more, see J. Bentley, *Programming Pearls*, Addison Wesley, 1986