1. Efficiency of Algorithms

Programming and Algorithms II
Degree in Bioinformatics
Fall 2017
What is a Good Program?

External:
• Correct – does what it’s supposed to do
• Efficient – uses few resources
  – Time, memory, bandwidth, disk...
• Robust – does not explode if something not quite right
• Easy to install, to use, to learn
• Goes along with other programs
• Works in many environments

• Internal
  – Easy to understand its code
  – Easy to modify - to do something different
  – Parts can be reused
Correct – does what it is supposed to do

• Think hard
• There are formal methods – won’t see them
• Testing:
  Can show that a program is incorrect
  Not that it is correct

• But what is it supposed do do?
  Always document your functions (programs)
Efficient – uses small resources

We focus on running time and memory

What is “to run fast”?  
1 sec? 1 minute? 1 hour? 1 week?

That depends on machine, interpreter, O.S.....

Also, we expect larger inputs to take more time
Efficient – uses small resources

We focus on running time and memory

What is “to run fast”? 1 sec? 1 minute? 1 hour? 1 week?

That depends on machine, interpreter, O.S. ...

Can we say something without executing it?
Efficient – uses small resources

Efficient = “it scales well to larger data”

• Time same for all inputs?
• x2 time if input size x2?
• x4 time if input size x2?
• Can only run for inputs < 10? (bad)
Orders of magnitude

Let \( f \) be a function from \( \mathbb{N} \) to \( \mathbb{N} \)

Definition of “Big-Oh”:

\[ O(f) = \{ g : \exists c \ \exists m \ \forall n > m \ g(n) < c \cdot f(n) \} \]

Equivalently, “\( g \) is in the order of \( f \)”

\[ g \in O(f) \iff \lim_{n \to \infty} \left( \frac{g(n)}{f(n)} \right) < \infty \]
Orders of magnitude

\[ 3n^2 \in O(0.01 \, n^2) \]

\[ 1000 \, n \in O(0.01 \, n^2) \]

\[ 3n^2 + 100 \, n + 1000 \in O(n^2) \]

\[ 20 \in O(1) \]

It’s wrong, but we write

\[ 3n^2 + 10 \, n + 100 = O(n^2) \]
$100n + 300$

$0.1n^2$
$100n + 300$

$0.1n^2$
Graph showing the growth of four functions:

- Red line: $10n + 30$
- Green line: $n^2$
- Yellow line: $0.1n^3$
- Blue line: $0.012^n$

The X-axis represents $n$, and the Y-axis represents the value of each function as $n$ increases.
Orders of magnitude: Usual ones

1
log(n)
$n^{1/2}$
n
$n \log(n)$
$n^2$
$n^3$
$2^n$
$3^n$
n! Stirling’s approximation $\approx (2\pi n)^{1/2} (n/e)^n$
Orders of magnitude: Scaling

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Linear: $f = O(n)$:</th>
<th>Quadratic $f = O(n^2)$:</th>
<th>Exponential $f = O(2^n)$:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$f(2n) \approx 2 \ f(n)$</td>
<td>$f(2n) \approx 4 \ f(n)$</td>
<td>$f(n+1) \approx 2 \ f(n)$</td>
</tr>
<tr>
<td></td>
<td>$f(10n) \approx 10 \ f(n)$</td>
<td>$f(10n) \approx 100 \ f(n)$</td>
<td>$f(n+10) \approx 1000 \ f(n)$</td>
</tr>
</tbody>
</table>
Worst-case analysis

We say

“the running time of algorithm A is $O(n)$”

if for every $n$

$max\{ \text{running time of } A(x) : x \text{ an input of size } n \} \text{ is } O(n)$

We take the worst-case input of every size
Bounding running time of programs

\[\text{time(simple expression)} = \text{“number of elementary instructions executed”}\]

\[\text{time}(I_1 \text{ then } I_2) = \text{time}(I_1) + \text{time}(I_2)\]

\[\text{time(if}(\text{cond}): I_1 \text{ else: } I_2) \leq \text{time}(\text{cond}) + \max(\text{time}(I_1), \text{time}(I_2))\]

\[\text{time}(\text{f}(\text{arg1, arg2, arg3})) = \text{time}(\text{arg1}) + \text{time}(\text{arg2}) + \text{time}(\text{arg3}) + \text{time}(\text{f}(\text{val1, val2, val3}))\]
Bounding running time of programs

\[ \text{time(while cond: body)} = \sum_i \text{time(cond}_i) + \text{time(body}_i) \]

where \( i \) ranges over the iteration number and \( \text{cond}_i \) and \( \text{body}_i \) are the executions of \( \text{cond} \) and \( \text{body} \) in the \( i \)th iterations.

If all iterations take roughly the same time, a good bound is:

\[ \text{time(while cond: body)} \leq (\text{number of iterations}) \times (\text{time(cond}) + \text{time(body})) \]
Example 1

```python
a = int(input("a: "))
b = int(input("b: 
"))
x = 3*a + 10
y = 9*b - 5
if (x > y):
    theMax = x
else:
    theMax = y
print(theMax)
```
Example 2

• Given a natural number \( n \), print all natural numbers that divide \( n \) exactly
• E.g., for 12 print 1, 2, 3, 4, 6, 12

```python
for i in range(1, n+1):
    if (n % i == 0): print(i)
```

• Value vs. number of digits
• Time \( \approx 10^{\text{size of the input}} \)
Example 3

```python
s = 0
for x in lst:
    s += x

s=0
for i in range(0,len(lst)):
    s += lst[i]

print(sum(lst))

print(sum(lst[i:j]))
```

$O(len(lst))$ $O(len(lst))$ $O(len(lst))$ $O(j-i)$
# Python lists (n=len(lst))

<table>
<thead>
<tr>
<th>Operation</th>
<th>Call</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>creating</td>
<td>lst = [v1,v2,...,vn]</td>
<td>O(n)</td>
</tr>
<tr>
<td>access by position</td>
<td>lst[i]</td>
<td>O(1)</td>
</tr>
<tr>
<td>append at end</td>
<td>lst.append(x) or</td>
<td>O(1) (on average...)</td>
</tr>
<tr>
<td></td>
<td>lst.extend(lst2)</td>
<td>O(len(lst2))</td>
</tr>
<tr>
<td>remove from end</td>
<td>lst.pop()</td>
<td>O(1) (on average...)</td>
</tr>
<tr>
<td>Insert at position</td>
<td>lst.insert(pos,x)</td>
<td>O(n)</td>
</tr>
<tr>
<td>delete at position</td>
<td>lst.pop(pos)</td>
<td>O(n)</td>
</tr>
<tr>
<td>count occurrences</td>
<td>l.count(x)</td>
<td>O(n)</td>
</tr>
<tr>
<td>make empty</td>
<td>l.clear()</td>
<td>O(1)</td>
</tr>
<tr>
<td>reverse</td>
<td>lst.reverse()</td>
<td>O(n)</td>
</tr>
<tr>
<td>copy</td>
<td>lst.copy()</td>
<td>O(n)</td>
</tr>
<tr>
<td>sort</td>
<td>lst.sort()</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>Iterate / membership</td>
<td>for x in lst  x in/not in lst</td>
<td>O(n)</td>
</tr>
<tr>
<td>copy slice</td>
<td>lst[a:b]</td>
<td>O(b-a)</td>
</tr>
<tr>
<td>check equality</td>
<td>lst1 == lst2</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Sorting a list

Given:
• a binary comparison operator “<“ (with what we expect from “<“... details omitted)
• a list whose elements are comparable with <

Do: permute the elements in L so that they are increasingly ordered w.r.t. <

[5,2,9,0,4]  \rightarrow [0,2,4,5,9]
[“dog”,”fish”,”ant”,”frog”]  \rightarrow
  [“ant”,”dog”,”fish”,”frog”]
Sorting a list

General rule: Don’t write your sorting algorithm

Almost always: list.sort()

But we’ll see some sorting algorithms:
• Selection
• Insertion
• Mergesort
• Quicksort
• Bubblesort simply don’t!
Selection sort


A LIST

98 is largest

87 is largest

72 is largest

66 is largest

52 is largest

51 is largest

36 is largest

29 is largest

sorting completed
def selectionSort(lst):
    for i in range(0, len(lst) - 1):
        posMin = i
        for pos in range(i + 1, len(lst)):
            if lst[pos] < lst[posMin]:
                posMin = pos
        lst[i], lst[posMin] = lst[posMin], lst[i]
def selectionSort(lst):
    for i in range(0, len(lst) - 1):
        posMin = i
        for pos in range(i + 1, len(lst)):
            if lst[pos] < lst[posMin]:
                posMin = pos
        lst[i], lst[posMin] = lst[posMin], lst[i]

Cost O(len(lst)-i)

If we call N=len(lst), cost is

O(N-2)+O(N-3)+O(N-4)+...+O(3)+O(2)+O(1)
Selection sort: Running time

Inner loop
- for pos in range(1,i+1): i iterations
- Body has $O(1)$ running time
- $\rightarrow O(i)$ running time

Outer Loop
- i in range 1..len(lst)-1 approx
- $O(1) + ... + O(len(lst))$ ?
Selection sort: Running time

\[ 1 + 2 + \ldots + (N-1) + N = \frac{N(N+1)}{2} \]

• By induction
• Or geometrically
Selection sort: Running time

Selection sort on a list of length $N$

• Running time $O(N^2)$

• $3(N-1)$ element assignments

• $N(N-1)/2$ element comparisons
Selection sort

Insertion sort

Assume 54 is a sorted list of 1 item

inserted 26

inserted 93

inserted 17

inserted 77

inserted 31

inserted 44

inserted 55

inserted 20
def insertionSort(lst):
    for i in range(1, len(lst)):
        x = lst[i]
        pos = i
        while pos > 0 and lst[pos-1] > x:
            lst[pos] = lst[pos-1]
            pos = pos-1
        lst[pos] = x

Cost O(i) – at most

If we call N=len(lst), cost is

O(1)+O(2)+O(3)+...+O(N-3)+O(N-2)+O(N-1)
Inner loop:
• Body is $O(1)$
• Executed for $j=0..i$
→ $O(i)$ running time

Outer loop:
• Executed for $i=0...\text{len(lst)}$
• Cost of $i$-th iteration $O(i)$
• $O(1) + ... + O(\text{len(lst)}) = O(\text{len(lst)}^2)$
Insertion sort

Insertion sort: Running time

Potentially $O(N^2)$ movements and comparisons

• If in reverse order

But faster when elements near their place (almost sorted): short inner loop!

Stable: “Equal” elements (neither $<$ nor $>$) remain in the same order as they were
To remember

Selection sort:
• + Easy to remember
• + O(N) movements
• - Not stable

Insertion sort:
• - Potentially O(N^2) movements
• + Faster if most elements near their place
• + But usually faster than selection sort
• + Stable
To remember

$O(\cdot)$ useful to discuss the running time of algorithms as data gets large

If two algorithms differ in $O(\cdot)$, one is much much faster than the other for large enough inputs

We know two algorithms that sort lists in time $O(n^2)$
Later we will see faster algorithms with time $O(n \log n)$
Some links

• Sorting algorithms
  https://www.youtube.com/watch?v=h-QYzgTmgVI

• Selection sort folk dance:
  https://www.youtube.com/watch?v=Ns4TPTC8whw

• Insertion sort folk dance:
  https://www.youtube.com/watch?v=ROalU379I3U