Lecture 16: Estimating Algorithm Performance

Dr Kieran T. Herley

Department of Computer Science
University College Cork

2013/14

Summary

Performance estimation as bean-counting exercise. Characteristic operations. Worst-case performance criterion. Analysis of simple algorithms.

Need for Efficient Algorithms

Typical Computational Problem Integer multiplication: take two n-bit binary integers and compute their product (n "large", $\gg 32$).

Goal Choose efficient (*i.e.* fast) algorithm.

Motivation Fast arithmetic important for encryption *etc.*

Some Multiplication. Algorithms Assume multiplicands X and Y are represented by bit arrays.

- **A** Add Y copies of integer X to form product
- **B** Primary-school, add-and-shift algorithm
- **C** "Clever" algorithm – details on web page

Comparing Algorithms

Running Times (ms)

n	C	B	A
4	1300	300	100
8	400	100	300
16	800	100	61600
32	1100	100	Huge
64	1700	400	Huge
128	3900	1400	Huge
256	11500	5200	Huge
512	35000	20200	Huge
1024	104300	80900	Huge

[a] Total times to multiply many pairs of randomly chosen operands.

Observation

A is effectively useless.

Comparing Algorithms

Running Times (ms)			
n	C	B	A
4	1300	300	100
8	400	100	300
16	800	100	61600
32	1100	100	Huge
64	1700	400	Huge
128	3900	1400	Huge
256	11500	5200	Huge
512	35000	20200	Huge
1024	104300	80900	Huge

[a]

[a] Total times to multiply many pairs of randomly chosen operands.

Observation

A is effectively useless.

At a minimum we would like to exclude algorithms with dreadful performance

Comparison cont'd

Running Times (ms)		
n	C	B
128	3900	1400
256	11500	5200
512	35000	20200
1024	104300	80900
2048	327900	323400
4096	968600	1280900
8192	2815300	5805400
16384	8741100	33787800

Observation

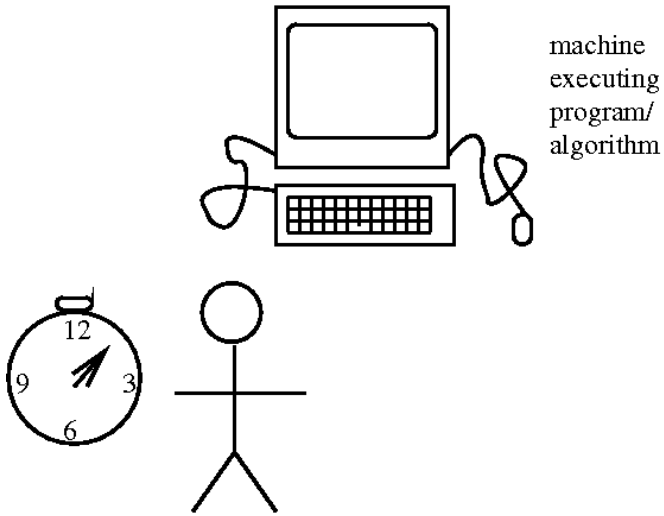
Algorithm C is more efficient than B when n is sufficiently large (> 2048) and this superiority becomes more marked the larger n gets.

Why Care About Large n?

- The growth rate of the algorithm's running time determines how well the algorithm "scales up" to larger inputs: n^2 algorithms scale up better than n^3 ones.
- Many tasks are computationally voracious and involve large data sets
- Examples
  - Scientific/engineering calculations
  - Multimedia applications
  - Modelling and forecasting
- Algorithm efficiency can determine viability of application

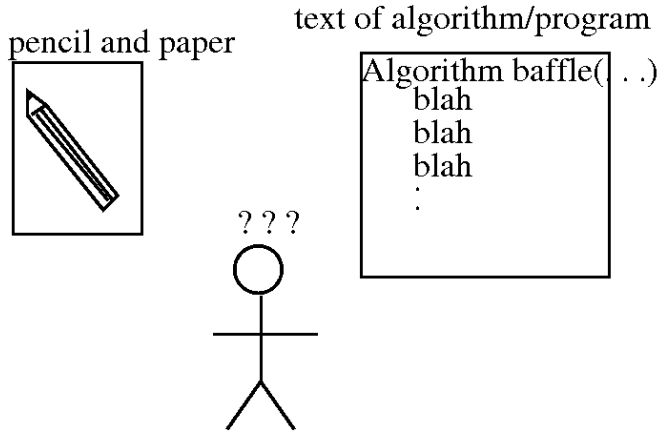
Assessing Algorithm Efficiency

Experimental Approach



Test program under "realistic conditions" and measure performance directly.

Analytical Approach–Our Choice



Analyse algorithm mathematically to derive formula characterising performance. Formula captures the number of *steps* (clarified later) executed when the algorithm is run.

Drawbacks with Experimental Approach

Idea: Test program under "realistic conditions" and measure performance directly.

Drawbacks:

a

[a] www.wikipedia.com

- Need to implement algorithm and "experimental apparatus".
- Can only test a few inputs. Which to choose and why?
- To compare algorithms, identical hardware/software needs to be used.

Characteristic Operations

Performance estimation as "bean-counting" exercise

- Estimate/count the number of *characteristic operations*(CO) that occur during algorithm's execution
- Choose COs so that
  - CO count reflective of total amount of work(*e.g.* bytecode steps) involved in executing algorithm
  - Relative CO-count of two algorithms reflects the relative performance

We choose *comparisons* ($\leq$ *etc.*) as our COs

Example 1

```
Algorithm comparisonExchange(A, a, b):
    if A[a] > A[b] then
        temp = A[a]
        A[a] = A[b]
        A[b] = temp
```

Example 1

```
Algorithm comparisonExchange(A, a, b):
    if A[a] > A[b] then
        temp = A[a]
        A[a] = A[b]
        A[b] = temp
```

Number of comparisons:

Example 1

```
Algorithm comparisonExchange(A, a, b):
    if A[a] > A[b] then
        temp = A[a]
        A[a] = A[b]
        A[b] = temp
```

Number of comparisons: 1

Example 2

```
Algorithm ArraySum(A, n):
    sum ←0
    for i ←0 to n−1 do
        sum ←sum + A[i]
    return sum
```

Note: n denotes length of A

Notational Aside

Regard for loops as while loops "in disguise".

for k $\leftarrow$s **to** f **do**
    /* loop body */

$\equiv$

k $\leftarrow$s
while k $\leq$ f **do**
    /* loop body */
    k $\leftarrow$k + 1

Example 2 cont'd

```
Algorithm ArraySum(A, n):
    sum ←0
    i ←0
    while i ≤n−1 do
        sum ←sum + A[i]
        i ←i + 1
    return sum
```

- # comparisons = # iterations +1
- # iterations

Example 2 cont'd

```
Algorithm ArraySum(A, n):
    sum ←0
    i ←0
    while i ≤n−1 do
        sum ←sum + A[i]
        i ←i + 1
    return sum
```

- # comparisons = # iterations +1
- # iterations = n
- # comparisons = $n + 1$

Example 2 cont'd

```
Algorithm ArraySum(A, n):
    sum ←0
    i ←0
    while i ≤n−1 do
        sum ←sum + A[i]
        i ←i + 1
    return sum
```

- # comparisons = # iterations +1
- # iterations = n
- # comparisons = $n + 1$

Observation

Our $n + 1$ running-time estimate captures the linear relationship between the array length n and the execution time: the fact that if we doubled the length of the array we would expect the running time also to double.

Example 3

Code

```
Algorithm ArrayMax(A, n)
    currentMax ←A[0]
    for i ←1 to n−1 do
        if currentMax < A[i] then
            currentMax ←A[i]
    return currentMax
```

Example 3 cont'd

```
(1) Algorithm ArrayMax(A, n)
(2)     currentMax ←A[0]
(3)     i ←1
(4)     while i ≤n−1 do
(5)         if currentMax < A[i] then
(6)             currentMax ←A[i]
(7)         i ←i +1
(8)     return currentMax
```

Example 3 cont'd

```
(1) Algorithm ArrayMax(A, n)
(2)     currentMax ←A[0]
(3)     i ←1
(4)     while i ≤n−1 do
(5)         if currentMax < A[i] then
(6)             currentMax ←A[i]
(7)         i ←i +1
(8)     return currentMax
```

- # iterations $= n - 1$
- # comparisons=

line	# comps
4	n
5	$n-1$
$\sum$	$2n-1$

Aside

- A more refined approach would consider a broader range of operations not just comparisons
- We limit ourselves to comparisons
  - for simplicity
  - because most of our examples will be "comparison-based"

Example 4

```
Algorithm ArrayFind(x, A, n):
    i ←0
    while i < n do
        if x = A[i] then
            return i
        else
            i ←i + 1
    return −1
```

- Number of comparisons: ???

Example 4

```
Algorithm ArrayFind(x, A, n):
    i ←0
    while i < n do
        if x = A[i] then
            return i
        else
            i ←i + 1
    return −1
```

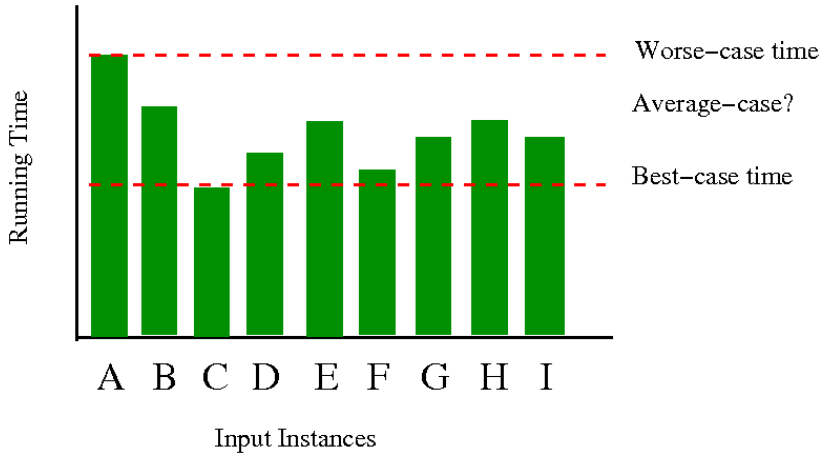
- Number of comparisons: depends on x and contents of A

Example 4

```
Algorithm ArrayFind(x, A, n):
    i ←0
    while i < n do
        if x = A[i] then
            return i
        else
            i ←i + 1
    return −1
```

- Number of comparisons: depends on x and contents of A
- There are many different *instances i.e.* values of A/x; precise number of comparisons varies
- What to do?

Worst-Case Running Time



- We use the *maximum* number of comparisons required among all input instances of "size" n as our estimate
- Here "size" refers to length of array
- Why?
  - Generally "easy" to determine
  - Provides reliable, conservative estimate of performance

Example 4 Again

```
Algorithm ArrayFind(x, A, n):
    i ←0
    while i < n do
        if x = A[i] then
            return i
        else
            i ←i + 1
    return −1
```

The *worst-case running time* is the maximum number of comparisons taken over all possible array contents of arrays of length n, expressed as a function of n.

Example cont'd

```
(1) Algorithm ArrayFind(x, A, n):
(2)     i ←0
(3)     while i < n do
(4)         if x = A[i] then
(5)             return i
(6)         else
(7)             i ←i + 1
(8)     return −1
```

Example cont'd

```
(1) Algorithm ArrayFind(x, A, n):
(2)     i ←0
(3)     while i < n do
(4)         if x = A[i] then
(5)             return i
(6)         else
(7)             i ←i + 1
(8)     return −1
```

- How many iterations?

Example cont'd

```
(1) Algorithm ArrayFind(x, A, n):
(2)     i ←0
(3)     while i < n do
(4)         if x = A[i] then
(5)             return i
(6)         else
(7)             i ←i + 1
(8)     return −1
```

- How many iterations? *at most n*
- Analysis

line	# comps
3	$\leq n+1$
4	$\leq n$

- Hence worst-case is $2n+1$

Notes

Worst-case analysis

- Provides "conservative" estimate of algorithm performance *e.g.* assurance that no execution of ArrayFind can take mnore than $2n + 1$ steps
- In many cases worst-case is reflective of "typical" performance

Other possibilities

- Could use *average-case performance* instead
- Tends to be more complicated to analyze
- Not necessarily a more "authentic" reflection of actual performance

Binary Search

```
Algorithm BinarySearch(S, k)
    low ←0
    high ←S.size() − 1
    while low ≤ high do
        mid = (low + high)/2
        midKey = key(mid)
        if k = midKey then
            return mid
        else
        if k < midKey then
            high ←mid −1
        else
            low ←mid+1
    return −1
```

Comparisons

- ifs: max two per iteration
- while: one per it. plus one
- total: $3\#\text{iterations} + 1$

Binary Search

```
Algorithm BinarySearch(S, k)
    low ←0
    high ←S.size() − 1
    while low ≤ high do
        mid = (low + high)/2
        midKey = key(mid)
        if k = midKey then
            return mid
        else
        if k < midKey then
            high ←mid −1
        else
            low ←mid+1
    return −1
```

Search Interval
S[low .. high] halves in length each iteration (Lecture 10)

Halving Property (Lecture 10)

Assumption array length a power of two

Claim Each iteration halves the search interval length

Why?

- $S[\ell..\text{mid}-1]$:

$$\underbrace{(\text{mid}-1)-\ell+1}_{\text{length of } S[\ell..\text{mid}-1]} = (\ell+h)/2-\ell \leq \underbrace{(h-\ell+1)/2}_{\text{half length of } S[\ell..h]}$$

(Recall that / denotes *integer* division)

- $S[\text{mid}+1..h]$: similar reasoning

Efficiency of Binary Search (Lecture 10)

- Each iteration halves the search interval length
-

stage	search interval
0	n
1	$n/2$
2	$n/4$
$\ldots$	$\ldots$
i	$n/2^i$
$\ldots$	$\ldots$
$\log_2 n$	1

  - There are $1 + \log_2 n$ iterations

Efficiency of Binary Search (Lecture 10)

- Each iteration halves the search interval length
-

stage	search interval
0	n
1	$n/2$
2	$n/4$
$\ldots$	$\ldots$
i	$n/2^i$
$\ldots$	$\ldots$
$\log_2 n$	1

  - There are $1 + \log_2 n$ iterations
    - $\log_2 n$ halvings to reduce search interval size to 1
    - one further iteration to finish
  - # comparisons is $4 + 3 \log_2 n$ (worst case)
-