CS107
Introduction to Computer Science

Lecture 7, 8

An Introduction to Algorithms:

Efficiency of algorithms

Comparing Algorithms

- Algorithm
  - Design
  - Correctness
  - Efficiency
  - Also, clarity, elegance, ease of understanding
- There are many ways to solve a problem
  - Conceptually
  - Also different ways to write pseudocode for the same conceptual idea
- How to compare algorithms?

Efficiency of Algorithms

- Algorithms are implemented on **real** machines, which have limited resources
- Efficiency: Amount of resources used by an algorithm
  - Space (number of variables)
  - Time (number of instructions)
- When designing an algorithm must be aware of its use of resources
- If there is a choice, pick the more efficient algorithm!

Efficiency of Algorithms

Usually efficiency means time-efficiency, that is, running time.

Analyzing efficiency comes down to: faster is better.

How to measure/estimate time efficiency of an algorithm?

- let it run and see how long it takes
  - We don't want to implement it…
  - Also…
    - On what machine?
    - On what inputs?
    - On what size of input?

Time Efficiency

..depends on input

- Example: the sequential search algorithm
  - In the best case, how fast can the algorithm terminate?
    - Target is the first element
  - In the worst case, how fast can the algorithm terminate?
    - Target is the last element, or not in the list
- Example: What is the best-case of binary search?
  - Target is the middle element
- If the **best-case running times** of two algorithms are the same... Do we know which one is more efficient in general??? Nope.
  - We normally look at the **worst-case running time**, i.e., the longest it could possibly take for an input of a fixed size

Time efficiency

..depends on size of input

- Example: list is 3 5
  - Search: worst case 2 comparisons
  - Binary search: worst-case 2 comparisons
  - Does this mean they are equal?? Nope.
- We are interested in running time for **large values of the input**
- The differences between algorithms become larger as the input becomes larger
- Example: list of size 15
  - Search worst case is: 15 comparisons
  - Binary search worst case is 4 comparisons
- Running time is a function of the input size
  - usually the input size is denoted n
  - the running time will be a function of n

Time Efficiency

- We want a measure of time efficiency which is independent of machine, speed etc
- Basically, we want to be able to look at 2 algorithms in pseudocode and compare them (without implementing them)

- (Time) Efficiency of an algorithm:
  - assume ideal computer on which all instructions take the same amount of time
  - Efficiency = the number of instructions executed

- Is this accurate?
  - Not all instructions take the same amount of time…
  - But..it is a good approximation of running time in most cases

Time efficiency

Assume the input has size n.

We are normally interested in best-case and worst-case efficiency:

worst case efficiency

is the *maximum* number of instructions that an algorithm can take for *any* input of size n.

best case efficiency

is the *minimum* number of instructions that an algorithm can take for *any* input of size n.

Sometimes we are also interested in

average case efficiency

-the efficiency averaged on all possible inputs of size n

- must assume a distribution of the input

- we normally assume uniform distribution (all inputs are equally probable)

(Sequential) Search

- Variables: i, target, list a of 100 elements
- i=1
- while (i <= 100)
  - Print "enter number: " i ":"
  - Get a_i
  - i = i+1
- Get target
- Set found = 0, i = 1
- While ((i <= 100) and (found == 0))
  - If a_i == target set found = 1
  - i = i+1
- If (found ==1) print "Target found at position" i-1
- Else print " Target not found."

Analysis of Sequential Search

Assume the size of input list is n.

- Reading the n inputs from user:
  - 4n instructions
- The search loop
  - Best-case, target is found immediately: 3 instructions or so
  - Worst-case, target is not found: 3n + 1 instructions

- Thus, overall:
  - best case: 4n+3
  - worst-case: 7n+1

- Both cases are of the form an+b, i.e. linear in n

Order of Magnitude

- sequential search: 7n+1 instructions worst-case
  - Are these constants accurate? Can we ignore them?

- Simplification:
  - **ignore the constants, look only at the order of magnitude**
  - n, 0.5n, 2n, 4n, 3n+5, 2n+100, 0.1n+3 ….are all linear
  - we say that their order of magnitude is n, denoted as $\Theta(n)$
    - 3n+5 has order of magnitude n: $3n+5 = \Theta(n)$
    - 2n +100 has order of magnitude n: $2n+100=\Theta(n)$
    - 0.1n+3 has order of magnitude n: $0.1n+3=\Theta(n)$
    - ….

Analysis of binary search

Assume the size of the input list is n.

- Assume the input has been read from the user, i.e. we only look at the while loop that searches for the target.
- What is the best case?
  - Target is the middle element: some constant number of instructions
- What is the worst case?
  - Initially the size of the list in n
  - After the first iteration through the repeat loop, if not found, then either start = m or end = m ==> size of the list on which we search is n/2
  - Every time in the repeat loop the size of the list is halved: n, n/2, n/4,….
  - How many times can a number be halved before it reaches 1?

$\log_2 x$

- $\log_2 x$
  - The number of times you can half a (positive) number x before it goes below 1
  - Examples:
    - $\log_2 16 = 4$ [16/2=8, 8/2=4, 4/2=2, 2/2=1]
- $\log_2 n = m \Longleftrightarrow 2^m = n$
    - $\log_2 8 = 3 \Longleftrightarrow 2^3=8$

$\log_2 x$

Increases very slowly

- $\log_2 8 = 3$
- $\log_2 32 = 5$
- $\log_2 128 = 7$
- $\log_2 1024 = 10$
- $\log_2 1000000 = 20$
- $\log_2 1000000000 = 30$
- ...

Order Θ(log n)

- The search loop in binary search takes:
  - Best-case: constant number of instructions
  - Worst-case: 4 log n + 1 instructions or so
- We'll ignore the constants and call this order of magnitude Θ(log n)
- Examples:
  - 2 log n +30 has order Θ(log n)
  - log n + 2 has order Θ(log n)
  - 10 log n + 1 has order Θ(log n)

Comparing Θ(lg n) and Θ(n)

Note: Θ(1) means constant time

$\Theta(1) \ll \Theta(\lg n) \ll \Theta(n)$

Does efficiency matter?

Example:

- Say $n = 10^9$ (1 billion elements)
- 10 MHz computer ==> 1 instr takes 10^{-7} sec
  - Sequential search would take
    - $\Theta(n) = 10^9 \times 10^{-7}$ sec = 100 sec
  - Binary search would take
    - $\Theta(\lg n) = \lg 10^9 \times 10^{-7}$ sec = 30 $\times 10^{-7}$ sec = 3 microsec

Exercises

What is the efficiency of the following algorithm? Give a theta-expression for it.

```
Variables: i, n, list a of size 100
n = 100
Print "Enter " n "elements: "
i = 1
while (i <= n)
   print "enter next element"
   get a_i
   i = i+1
Print "Great, thanks."
```

Exercises

What is the efficiency of the following algorithms? Give a theta-expression for it. Distinguish between best and worst cases, if applicable.

- Computing the sum of all elements in a list of size n
- Computing the smallest or the largest number in a list of size n

More examples

- What is the efficiency of the following algorithm? Give a theta-expression for it.

- Get n
- I = 1
- while (I <= n)
  - print "*"
  - i = i+1
- j = 1
- while (j <= n)
  - print "-"
  - j = j+1

More examples

- Find the running time as a function of n for the following algorithm. It is enough to give a theta-expression for it.

- Get n
- i = 1
- while (i <= n)
  - print "****"
  - j = 1
  - while (j <= n)
    - print "j"
    - j = j+1
  - i = i+1
- Print "done"

Order of magnitude $\Theta(n^2)$

- Any algorithm that does on the order of cn^2 work for any constant c
  - $2n^2$ has order of magnitude $\Theta(n^2)$
  - $.5n^2$ has order of magnitude $\Theta(n^2)$
  - $100n^2$ has order of magnitude $\Theta(n^2)$
  - $10n^2 + 10n + 5$ has order of magnitude $\Theta(n^2)$
  - $3n^2 + 2n + 1$ has order of magnitude $\Theta(n^2)$

Orders of magnitude

- Comparing order of magnitudes

  $\Theta(1) << \Theta(\lg n) << \Theta(n) << \Theta(n^2)$

- There are other orders of magnitude, for instance $\Theta(n^3)$, $\Theta(n^4)$, $\Theta(n \log n)$, $\Theta(2^n)$, etc

- Problems which take $\Theta(2^n)$ time are called exponential. $\Theta(2^n)$ grows so fast with n that these problems are practically impossible to solve for n >15.

Comparison of $\Theta(n)$ and $\Theta(n^2)$

- $\Theta(n)$: n, 2n+5, 0.01n, 100n, 3n+10,..
- $\Theta(n^2)$: n^2, $10n^2$, $0.01n^2$, n^2+3n, n^2+10,…
- We do not distinguish between constants..
  - Then…why do we distinguish between n and n^2 ??
  - Compare the shapes: n^2 grows much faster than n
    - Anything that is order of magnitude n^2 will eventually be larger than anything that is of order n, no matter what the constant factors are
    - Fundamentally n^2 is more time consuming than n
  - $\Theta(n^2)$ is larger (less efficient) than $\Theta(n)$
    - $0.1n^2$ is larger than 10n (for large enough n)
    - $0.0001n^2$ is larger than 1000n (for large enough n)

The Tortoise and the Hare

Does algorithm efficiency matter??

- …**just buy a faster machine!**

Example:

- Apple desktop
  - 1GHz (10^9 instr per second), $2000
- Cray computer
  - 10000 GHz(10^{13} instr per second), $30million

- Run a $\Theta(n)$ algorithm on an Apple
- Run a $\Theta(n^2)$ algorithm on a Cray
- For what values of n is the Apple desktop faster?

Exercise

- Write an algorithm to ask the user for a number n and print a multiplication table with n lines and columns. In line I and column J it computes the value I*J.

  For example, for n=5

1x1 1x2 1x3 1x4 1x5
2x1 2x2 2x3 2x4 2x5
3x1 3x2 3x3 3x4 3x5
4x1 4x2 4x3 4x4 4x5
5x1 5x2 5x3 5x4 5x5

What is the running time of the algorithm, as a function of n?

Sorting

- Problem: sort a list of items into order
- Why sorting?
  - Sorting is ubiquitous (very common)!!
  - Examples:
    - Registrar: Sort students by name or by id or by department
    - Post Office: Sort mail by address
    - Bank: Sort transactions by time or customer name or accound number …
- For simplicity, assume input is a list of n numbers
  - The same ideas can be used to sort names, text, etc
- Ideas for sorting?

Selection Sort

- Idea: grow a sorted subsection of the list from the back to the front

5 7 2 1 6 4 8 3 |
5 7 2 1 6 4 3 | 8
5 2 1 6 4 3 | 7 8
5 2 1 3 4 | 6 7 8
…
| 1 2 3 4 5 6 7 8

Selection Sort

- When we try to solve a new problem, we start at a high level of abstraction, then go and fill in the details
- Selection sort, at a high level of abstraction
  - Get values for the list of n items
  - Set marker for the unsorted section at the end of the list
  - Repeat until unsorted section is empty
    - Select the largest number in the unsorted section of the list
    - Exchange this number with the last number in unsorted section of list
    - Move the marker of the unsorted section forward one position
  - End

Levels of abstraction

- It is easier to start thinking of a problem at a high level of abstraction
- Algorithms as building blocks
  - We can build an algorithm from "parts" consisting of algorithms which we already know
  - Selection sort:
    - Iterate through a loop
    - Select largest number in the unsorted section of the list
      - We have seen an algorithm to do this last time
    - Exchange 2 values in a list

Selection Sort

- One further step
  - Get $a_1, a_2, \ldots, a_n$
  - Set unsortedEnd = n
  - While (unsorted > 1)
    - Find the position of the largest element in the unsorted section of the list, that is, among $a_1, a_2, \ldots, a_{unsortedEnd}$
    - Assign this position to p
    - Swap a_p with $a_{unsortedend}$
    - unsortedEnd = unsortedEnd -1
  - End

Selection Sort Analysis

- Iteration 1:
  - Find largest value in a list of n numbers : n-1 comparisons
  - Exchange values and move marker
- Iteration 2:
  - Find largest value in a list of n-1 numbers: n-2 comparisons
  - Exchange values and move marker
- Iteration 3:
  - Find largest value in a list of n-2 numbers: n-3 comparisons
  - Exchange values and move marker
- …
- Iteration n:
  - Find largest value in a list of 1 numbers: 0 comparisons
  - Exchange values and move marker

Total: (n-1) + (n-2) + …. + 2 + 1

Selection Sort

- Total work (nb of comparisons):
  - (n-1) + (n-2) + …. + 2 + 1
  - This sum is equal to $.5n^2 -.5n$ (proved by Gauss)

  => order of magnitude is $\Theta(n^2)$
- Questions
  - best-case, worst-case ?
- Other sorting ideas?
- Can we find more efficient sorting algorithms? That is, faster than $\Theta(n^2)$ in the worst case?

Efficiency of algorithms

- Summary: count the number of instructions ignoring constants
- order of magnitudes

  $\Theta(1) << \Theta(\lg n) << \Theta(n) << \Theta(n^2)$
- We cannot compare two algorithms in the same class
  - if we want to do this, we need to count carefully the constants, among others.
- but we know that a running time of $\Theta(n)$ is faster than $\Theta(n^2)$, for large enough values of n
- If algorithm1 has a worst-case efficiency of $\Theta(n)$ and algorithm2 has a worst-case efficiency of $\Theta(n^2)$, then algorithm1 is faster (more efficient) in the worst-case
- At the algorithm design level we want to find the most efficient algorithm in terms of order of magnitude
- We can worry about optimizing each step at the implementation level