Algorithms

Overview

Algorithms

- A generic recipe for computation
    - Finite sequence of instructions
        - to solve a computational problem
- Should work on broad category of computers
    - E.g. Algorithms for quantum computers, biological computers are / would be different

Algorithms

- Correctness
  - Is the algorithm *guaranteed* to give the correct result
  - Is the algorithm guaranteed to give the correct result using resources
- Performance
  - Measured in resource use:
    - Time
    - Storage / memory
    - Energy use
    - Overwrites in an SSD, …

Standard Model of Computing

- What is presented to the programmer:
  - Computer reads instructions from memory
  - Computer acts on instructions by changing memory locations
    - Example: addi x, 5
      - Load x into accumulator, load 5 into a register, add results, move accumulator results back into memory where x is located

Standard Model of Computing is an Idealization

- Instructions do not take the same amount of time
    - Almost since the beginning of computer architecture
    - Idealization: Fetch-Execute Cycle with fixed timing
- Instructions are not performed serially
    - Pipelining of instructions
    - Reordering of instructions by compiler or architecture

Standard Model of Computing is an Idealization

- Memory access is not uniform
  - Early modification: Virtual Memory

Standard Model of Computing is an Idealization

- Modern modification:
  - Registers
  - Cache Level 1
  - Cache Level 2
  - Cache Level 3
  - Main Memory (DRAM)
  - Storage
    - Buffer - Cache - HDD block / SSD page

Standard Model of Computing is an Idealization

- Multi-threaded (e.g. multi-core) :
  - Many instructions & access to variables are not thread-safe
    - E.g.: Can only argue that a flag is either set or not if the flag is "atomic" (with software and hardware support)
  - Multi-core architecture manages to prevent a processor from having a different view of memory than another processor
    - But this is getting more and more difficult

Standard Model of Computing is an Idealization

- Storage and Memory systems prioritize reads over writes
- In case of failure, bad things can happen:
  - Can store a block
  - Read from this block
  - Power failure
  - Read from the block:
    - Value has changed

Standard Model of Computing

- Contract between system and programmer:
  - System does what programmer wants, but in a different faster way
  - With a few exceptions, which makes multi-threaded computing so challenging

Standard Model of Computing

- Turns out that the optimizations of modern computing systems **do not** create genuine new capabilities
- We can ***emulate*** a modern system using an old one
- We can even ***emulate*** a modern system using a model of computing used in the 30s and 40s to model what Mathematics can compute:
  - Turing machine

DNA Computing

- DNA can store vast amounts of information in a very small space.
  - Store data (key-value pair) by encoding in DNA sub-sequences
  - To look up by key:
    - Introduce the compliment of the key's substring affixed to a magnetic bead
    - Compliment bonds to DNA molecules with that key
    - Extract these DNA molecules magnetically
    - Sequence them for the result
- Does not change the basic capabilities

Quantum Computing

- Uses quantum phenomena for computing
    - Especially super-position and entanglement
  - Can be analog or digital
  - Digital quantum computing uses quantum gates
  - Difficulty now is getting up the number of q-bits in a system
- Could be faster than classical computers
  - Example: Shor's algorithm for factoring integers, Boson sampling
- Will almost certainly force current cryptography to use much larger keys

Quantum Computing

- Does not seem to change what is computable
- Changes possibly dramatically the speed at which things can be computed

Algorithms

- Algorithms $\neq$ Implementation
    - An algorithm can be implemented more or less efficiently
    - You can measure the speed of an implementation on a given system fairly accurately
    - You can derive the performance of an algorithm using a computing model

Algorithms

- Correctness
  - Can we prove that the answer given by an algorithm is correct?
    - via Automated proof methods
    - via human reasoning
  - Often involves pseudo-code

Algorithms

- Performance
  - Needs to be measured independently of implementation
  - Depends on the "instance size"
    - Many problems in CS become proportionally <u>more difficult</u> as they grow
    - Use an "asymptotic" notation to capture behavior as we "scale up"

Performance

- Computing uses resources
  - Space: How much storage is needed
  - Time: How many instructions are needed
- But it becomes more interesting:
  - Some problems need to use storage (flash / disks)
    - Storage is much slower
    - Performance measurement: How many times does the algorithm need to access storage

Performance

- Parallel / Multi-threaded performance
  - Almost all computers have limited capability to execute instructions in parallel
  - E.g.: Develop data structures that are
    - thread-safe
    - lock-free (no locking of shared resources needed)
    - wait-free (no waiting for a thread to access a data structure)

Impossibility Results

- Can all problems be solved with a computer
    - Depends on the type of computer, but:
        - In a very generic computing model, there are problems that cannot be solved

Impossibility Results

- Are there problems that can become prohibitively expensive?
- Answer: Probably yes. There are classes of problems that become intractable as they scale up

Outlay of Class

- Goal:
  - You are to develop the capability to argue about the
    - correctness
    - performance
  - of algorithms and data structures
  - You are to develop the capability to invent simple algorithms and data structures
  - You are to develop the capability to implement algorithms and data structures

Outlay of Class

- Contents:
    - Finite automata and regular expressions
    - Recurrence, asymptotic comparisons, and divide-and-conquer problems
    - Fast Data Structures
    - Dynamic and greedy programming
    - Graph Algorithms
    - Limits of Computability
    - Complexity Classes

Introduction to Performance

Counting Operations

- Type 1: All operations take the same time
- Type 2: Only count certain operations
- Type 3: Count how often the instructions in the body of a loop are executed

Counting Operations

- Example 1:

```
def bubble_sort(an_ar):
    for i in range(len(an_ar)):
        for j in range(i+1, len(an_ar)):
            if an_ar[i] > an_ar[j]:
                an_ar[i], an_ar[j] = an_ar[j], an_ar[i]
```

Counting Operations

```
def bubble_sort(an_ar):
    for i in range(len(an_ar)):
        for j in range(i+1, len(an_ar)):
            if an_ar[i] > an_ar[j]:
                an_ar[i], an_ar[j] = an_ar[j], an_ar[i]
```

- Identify the inner loop

Counting Operations

```
def bubble_sort(an_ar):
    for i in range(len(an_ar)):
        for j in range(i+1, len(an_ar)):
            if an_ar[i] > an_ar[j]:
                an_ar[i], an_ar[j] = an_ar[j], an_ar[i]
```

- Count the number of times the inner loop is executed

Counting Operations

```
def bubble_sort(an_ar):
    for i in range(len(an_ar)):
        for j in range(i+1, len(an_ar)):
            if an_ar[i] > an_ar[j]:
                an_ar[i], an_ar[j] = an_ar[j], an_ar[i]
```

- Let n be the number of elements in the array

Counting Operations

```
def bubble_sort(an_ar):
    for i in range(len(an_ar)):
        for j in range(i+1, len(an_ar)):
            if an_ar[i] > an_ar[j]:
                an_ar[i], an_ar[j] = an_ar[j], an_ar[i]
```

- Let n be the number of elements in the array
    - For i = 0: `for j in range(1, n):`
        - $n - 1$ repetitions

Counting Operations

```
def bubble_sort(an_ar):
    for i in range(len(an_ar)):
        for j in range(i+1, len(an_ar)):
            if an_ar[i] > an_ar[j]:
                an_ar[i], an_ar[j] = an_ar[j], an_ar[i]
```

- Let n be the number of elements in the array
    - For i = 0: $n-1$ repetitions
    - For $i=1$: $n-2$ repetitions
    - …
    - For $i=n-1$: 0 repetitions

Counting Operations

```
def bubble_sort(an_ar):
    for i in range(len(an_ar)):
        for j in range(i+1, len(an_ar)):
            if an_ar[i] > an_ar[j]:
                an_ar[i], an_ar[j] = an_ar[j], an_ar[i]
```

- Let n be the number of elements in the array
  - $(n-1) + (n-2) + \ldots + 2 + 1 + 0$ repetitions
  - i.e. $\dfrac{(n-1) \cdot n}{2}$ repetitions

Counting Operations

- Often, the input determines the number of operations
- Example: Quicksort
  - Recursive operations based on
    - select a random pivot
    - partition array around random pivot
    - quick-sort each partition
    - If partition are very small, use another sorting algorithms

Counting Operations

- Partition cost: *n* comparisons for *n* elements in array
- Best Case:
  - Pivots are always chosen to divide the array evenly
  - ((1+1+1)+1+(1+1+1))+1+((1+1+1)+1+(1+1+1))
  - Ideal array size is $2^{n+1} - 1$ with *n* steps
  - If elements are distinct:
    - Only one pivot
    - At each step, we have to partition all elements that were not pivot previously

Counting Operations

- Quicksort Best Case Performance:
  - $2^{n+1} - 1$ array elements with *n* steps
  - First step: Compare pivot with $2^{n+1} - 2$ elements
  - Second step: Previous pivot is no longer compared
    - Compares two pivots with a total of $2^{n+1} - 3$ elements
  - Third step: Previous pivots are no longer used
    - Compare a total of four pivots with a total of $2^{n+1} - 7$

Counting Operations

- Quicksort Best Case Performance:
  - $2^{n+1} - 1$ array elements with *n* steps
  - Total number of comparisons:
    - $n \cdot (2^{n+1} - 1) - (1 + 2 + 4 + \ldots + 2^{n-1})$

    $= 2^{n-1}(4n - 1) - n$
  - $N = 2^{n+1} - 1 \Longrightarrow n = \log_2(N + 1) - 1$
  - Total number of comparisons is
    - $1 - \log_2(N) + \dfrac{1}{4}(1 + N)(4(\log_2(N + 1) - 1)$

Counting Operations

- Quicksort Worst Case:
  - Pivot is always the smallest element
  - If there are N elements in the array:
    - $N-1$ rounds reducing the array by one element each time
    - $(N-1)+(N-2)+\ldots 1 = \dfrac{N(N-1)}{2}$ comparisons