Analysing Costs

Lecture 10

Menu

- Cost of operations and measuring efficiency
- ArrayList: retrieve, remove, add
- ArraySet: contains, remove, add

Analysing Costs

How can we determine the costs of a program?

- **Time**:
  - Run the **program** and count the milliseconds/minutes/days.
  - Count the number of steps/operations the **algorithm** will take.
- **Space**:
  - Measure the amount of memory the **program** occupies
  - Count the number of elementary data items the **algorithm** stores.
- Question:
  - Programs or Algorithms?
- Answer:
  - Both
    - programs: benchmarking
    - algorithms: analysis

Benchmarking: program cost

- Measure:
  - actual programs
  - on real machines
  - on specific input
  - measure elapsed time
    - **System.currentTimeMillis**()
      $\rightarrow$ time from system clock in milliseconds (long)
  - measure real memory usage
- Problems:
  - what input ⇒ choose test sets carefully
    use large data sets
    don't include user input
  - other users/processes ⇒ minimise
    average over many runs
  - which computer? ⇒ specify details

Analysis: Algorithm complexity

- Abstract away from the details of
  - the hardware
  - the operating system
  - the programming language
  - the compiler
  - the program
  - the specific input
- Measure number of “steps” as a function of the data size.
  - worst case (easier)
  - average case (harder)
  - best case (easy, but useless)
- Construct an expression for the number of steps:
  - cost = $3.47\ n^2 - 67n + 53$ steps
  - cost = $3n \log(n) - 5n + 6$ steps

  simplified into terms of different powers/functions of n

Analysis: Asymptotic Notation

- We only care about the cost when it is large
  - ⇒ drop the lower order terms
    (the ones that will be insignificant with large n)

    cost = $3.47\ n^2$ + ... steps

    cost = $3n \log(n)$ + … steps

- We don't care about the constant factors
  - Actual constant will depend on the hardware

  ⇒ Drop the constant factors

    cost $\propto n^2$ + … steps

    cost $\propto n \log(n)$ + … steps

- "Asymptotic cost", or "big-O" cost.
  - describes how cost grows with input size
  - cost is $O(1)$: fixed cost
  - cost is $O(n)$: grows with n

Big Oh Notation

- A notation for describing efficiency of computer algorithms
- assuming a 100-MHz clock, N = 1024k = 2^{20}

- O(1) - constant time, 10 ns

- O(log N) - logarithmic time, 200 ns
- O(N) - linear time, 10.5ms
- O(N log N) - n log n time, 210 ms

- O(N ^ 2) - quadratic time, 3.05 hours
- O(N ^ 3) - cubic time, 365 years

- O(2 ^ N) - exponential, 10 ^ (10 ^ 5) years

Typical Costs in Big 'O'

If data/input is size n		How does it grow.
O(1)	*"constant"*	cost is independent of *n*
O(log(n))	*"logarithmic"*	size x 10 $\rightarrow$ add a little (log(10)) to the cost
O(n)	*"linear"*	size x 10 $\rightarrow$ 10 x the cost
O(n log(n))	*"en-log-en"*	size x 10 $\rightarrow$ bit more than 10 x
O(n^2)	*"quadratic"*	size x 10 $\rightarrow$ 100 * the cost
O(n^3)	*"cubic"*	size x 10 $\rightarrow$ 1000 * the cost
:		
:		
O(2^n)	*"exponential"*	adding one to size -> doubles the cost => You don't want to run this algorithm!
O(n!)	*"factorial"*	adding one to size -> n the cost => You **definitely** don't want this algorithm!

Hierarchy of functions

- We can define a hierarchy of functions each having a **greater** order of magnitude than its predecessor:

constant	logarithmic	polynomial	exponential	
1	$\log n$	$n \quad n^2 \quad n^3 \ \ldots \ n^k \ \ldots$	2^n	$n!$

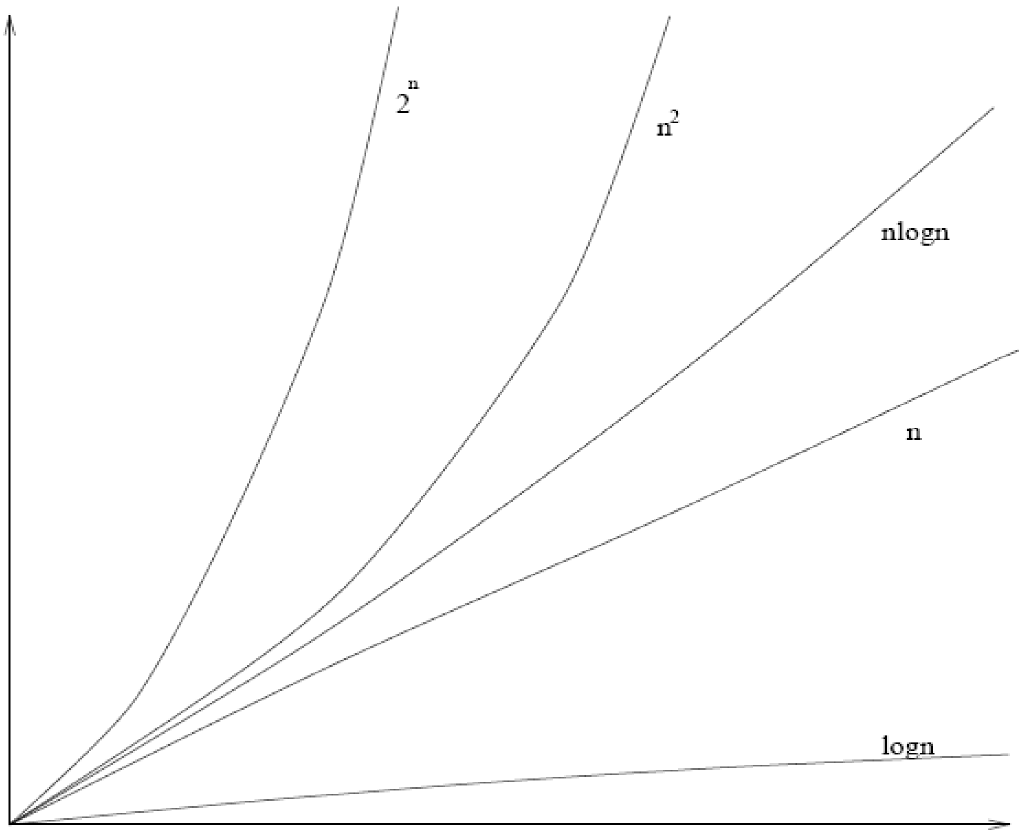
- We can further refine the hierarchy by inserting $\mathbf{n \log n}$ between $\mathbf{n}$ and $\mathbf{n^2}$, $\mathbf{n^2 \log n}$ between $\mathbf{n^2}$ and $\mathbf{n^3}$, and so on.

Which one is the fastest?

- Usually we are only interested in the ***asymptotic*** time complexity, i.e., when n is large

$O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

Typical Costs



Problem: What is a "step"?

- Count
  - actions in the innermost loop (happen the most times)
  - actions that happen every time round (not inside an "if")
  - actions involving "data values" (rather than indexes)
  - representative actions (don't need every action)

```
public  E  remove (int index){
      if (index < 0 || index >= count) throw new ….Exception();
      E ans = data[index];
      for (int i=index+1; i< count; i++)      ← in the innermost loop
         data[i-1]=data[i];                   ←Key Step
       count--;
      data[count] = null;
      return ans;
    }
```

Each for loop:

1 comparison: i<count

1 addition: i++

1 data retrieval: data[i]

1 subtraction: i-1

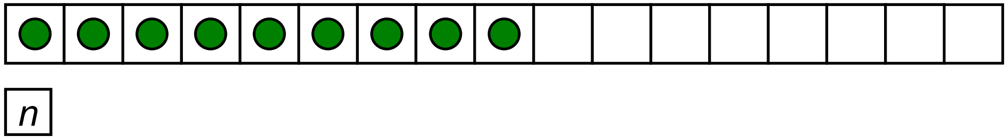
1 memory store: data[i-1]=data[i]

What's a step: Pragmatics

- Count the most expensive actions:
  - Adding 2 numbers is cheap
  - Raising to a power is not so cheap
  - Comparing 2 strings *may* be expensive
  - Reading a line from a file *may* be very expensive
  - Waiting for input from a user or another program may take forever…
- Sometimes we need to know how the underlying operations are implemented in the computer to analyse well

ArrayList: get, set, remove

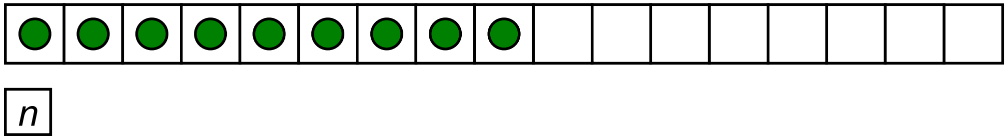
- Assume List contains *n* items.
- Cost of get and set:
  - best, worst, average: O(1)
  - ⇒ constant number of steps, regardless of *n*



- Cost of Remove:
  - worst case:
    - what is the worst case?
    - how many steps?
  - average case:
    - what is the average case?
    - how many steps?

ArrayList: add

- Cost of add(index, value):
  - what's the key step?
  - worst case:

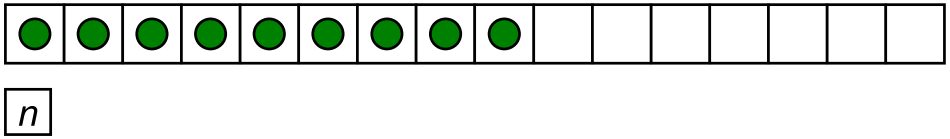


  - average case:

```
public void add(int index, E item){
  if (index<0 || index>count) throw new IndexOutOfBoundsException();
  ensureCapacity();
  for (int i=count; i > index; i--)
       data[i]=data[i-1];
  data[index]=item;
  count++;
}
```

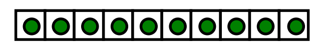
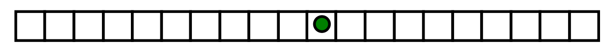
ArrayList: add at end

- Cost of add(value):
  - what's the key step?
  - worst case:
  - average case:




```
public void add (E item){
    ensureCapacity();
    data[count++] = item;
}
private void ensureCapacity () {
    if (count < data.length) return;
    E [ ] newArray = (E[ ]) (new Object[data.length * 2]);
    for (int i = 0; i < count; i++)
        newArray[i] = data[i];
    data = newArray;
  }
```

ArrayList: add at end

- Average case:
  - average over all possible states and input. OR
  - amortised cost over time.
- Amortised cost: total cost of adding *n* items ÷ *n* :
  - first 10: cost = 1 each total = 10 
  - 11th: cost = 10+1 total = 21 
  - 12-20: cost = 1 each total = 30
  - 21st: cost = 20+1 total = 51
  - 22-40: cost = 1 each total = 70
  - 41st: cost = 40+1 total = 111
  - 42-80: cost = 1 each total = 150
  - :
  - \- *n* total =
- Amortised cost (per item) =

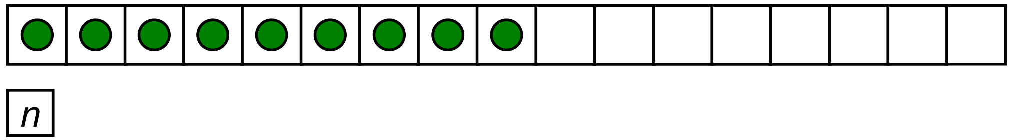
ArrayList costs: Summary

• get	O(1)	
• set	O(1)	
• remove	O(n)	
• add (at i)	O(n)	(worst and average)
• add (at end)	O(1)	(average)
	O(n)	(worst)
	O(1)	(amortised average)

- Question:
  - what would the amortised cost be if the array size is increased by 10 each time?

Cost of ArraySet

- ArraySet uses same data structure as ArrayList
  - does not need to keep items in order



- Operations are:
  - contains(item)
  - add(item) ← always add at the end
  - remove(item) ← don't need to shift down – just move last item down

- What are the costs?
  - contains: O(n)
  - remove:
  - add: O(1) O(n)

Q&A

- O(log(n)) < O(sqrt(n)) (T or F)
- O(n^n) < O(n!) (T or F)
- O(2^n) < O(n^n) (T or F)
- When analysing the cost of an algorithm, loop usually is the focus. (T or F)
- Which of the following operations is more expensive?
  - Reading a line from a file
  - Reading a line from a user
- Worst case cost analysis is usually more difficult than average cost analysis. (T or F)

Summary

- Cost of operations and measuring efficiency
- ArrayList: retrieve, remove, add
- ArraySet: contains, remove, add

Readings

- [Mar07] Read 2.2, 2.3, 2.4
- [Mar13] Read 2.2, 2.3, 2.4