Algorithm Analysis

Part II

Tyler Moore

CS 2123, The University of Tulsa

Some slides created by or adapted from Dr. Kevin Wayne. For more information see

`http://www.cs.princeton.edu/~wayne/kleinberg-tardos`.

Some slides adapted from Dr. Steven Skiena. For more information see `http://www.algorist.com`

Why it matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Implications of dominance

- Exponential algorithms get hopeless fast.
- Quadratic algorithms get hopeless at or before 1,000,000.
- $O(n \log n)$ is possible to about one billion.

Testing dominance

Definition

Dominance $g(n)$ dominates $f(n)$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Definition

Little oh notation $f(n)$ is $o(g(n))$ iff $g(n)$ dominates $f(n)$.

- In other words, little oh means "grows strictly slower than".
- Q: is $3n$ $o(n^2)$?
- A: Yes, since $\lim_{n\to\infty} \frac{3n}{n^2} = \frac{3}{n} = 0$
- Q: is $3n^2$ $o(n^2)$?
- A:

Useful facts

Proposition. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$, then $f(n)$ is $\Theta(g(n))$.

Pf. By definition of the limit, there exists n_0 such such that for all $n \geq n_0$

$$\frac{1}{2}c < \frac{f(n)}{g(n)} < 2c$$

- Thus, $f(n) \leq 2\,c\,g(n)$ for all $n \geq n_0$, which implies $f(n)$ is $O(g(n))$.
- Similarly, $f(n) \geq \frac{1}{2}\,c\,g(n)$ for all $n \geq n_0$, which implies $f(n)$ is $\Omega(g(n))$.

Proposition. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f(n)$ is $O(g(n))$.

Asymptotic bounds for some common functions

Polynomials. Let $T(n) = a_0 + a_1 n + \ldots + a_d n^d$ with $a_d > 0$. Then, $T(n)$ is $\Theta(n^d)$.

Pf. $\lim_{n\to\infty} \frac{a_0 + a_1 n + \ldots + a_d n^d}{n^d} = a_d > 0$

Logarithms. $\Theta(\log_a n)$ is $\Theta(\log_b n)$ for any constants $a, b > 0$. ← no need to specify base (assuming it is a constant)

Logarithms and polynomials. For every $d > 0$, $\log n$ is $O(n^d)$.

Exponentials and polynomials. For every $r > 1$ and every $d > 0$, n^d is $O(r^n)$.

Pf. $\lim_{n\to\infty} \frac{n^d}{r^n} = 0$

Exercises

- Using the limit formula and results from earlier slides, answer the following:
- Q: Is $5n^2 + 3n$ $o(n)$?
- A: No, since $\lim_{n\to\infty} \frac{5n^2+3n}{n} = \lim_{n\to\infty} 5n + 3 = \infty$
- Q: is $3n^3 + 5$ $\Theta(n^3)$?
- A:
- Q: is $n \log n + n^2$ $O(n^3)$?
- A:

Linear time: O(n)

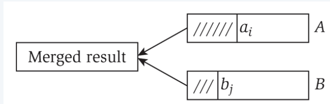
Linear time. Running time is proportional to input size.

Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max ← a1
for i = 2 to n {
   if (ai > max)
      max ← ai
}
```

Linear time: O(n)

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.




```
i = 1, j = 1
while (both lists are nonempty) {
   if (a_i ≤ b_j) append a_i to output list and increment i
   else         append b_j to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes $O(n)$ time.
Pf. After each compare, the length of output list increases by 1.

Linearithmic time: O(n log n)

O(n log n) time. Arises in divide-and-conquer algorithms.

Sorting. Mergesort and heapsort are sorting algorithms that perform $O(n \log n)$ compares.

Largest empty interval. Given n time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval when no copies of file arrive?

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Quadratic time: O(n²)

Ex. Enumerate all pairs of elements.

Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

O(n²) solution. Try all pairs of points.

```
min ← (x_1 - x_2)² + (y_1 - y_2)²
for i = 1 to n {
   for j = i+1 to n {
      d ← (x_i - x_j)² + (y_i - y_j)²
      if (d < min)
         min ← d
   }
}
```

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion. [see Chapter 5]

Cubic time: O(n³)

Cubic time. Enumerate all triples of elements.

Set disjointness. Given n sets $S_1, \ldots, S_n$ each of which is a subset of $1, 2, \ldots, n$, is there some pair of these which are disjoint?

O(n³) solution. For each pair of sets, determine if they are disjoint.

```
foreach set S_i {
   foreach other set S_j {
      foreach element p of S_i {
         determine whether p also belongs to S_j
      }
      if (no element of S_i belongs to S_j)
         report that S_i and S_j are disjoint
   }
}
```

Polynomial time: $O(n^k)$

Independent set of size k. Given a graph, are there k nodes such that no two are joined by an edge?

k is a constant

$O(n^k)$ solution. Enumerate all subsets of k nodes.

```
foreach subset S of k nodes {
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

- Check whether S is an independent set takes $O(k^2)$ time.
- Number of k element subsets = $\binom{n}{k} = \frac{n(n-1)(n-2)\times\cdots\times(n-k+1)}{k(k-1)(k-2)\times\cdots\times 1} \leq \frac{n^k}{k!}$
- $O(k^2\ n^k\ /\ k!) = O(n^k)$.

poly-time for k=17, but not practical

Exponential time

Independent set. Given a graph, what is maximum cardinality of an independent set?

$O(n^2\ 2^n)$ solution. Enumerate all subsets.

```
S* ← φ
foreach subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* ← S
   }
}
```


Sublinear time

Search in a sorted array. Given a sorted array A of n numbers, is a given number x in the array?

$O(\log n)$ solution. Binary search.

```
lo ← 1, hi ← n
while (lo ≤ hi) {
   mid ← (lo + hi) / 2
   if      (x < A[mid]) hi ← mid - 1
   else if (x > A[mid]) lo ← mid + 1
   else return yes
}
return no
```


Common algorithm dominance classes

Dominance class	Example problem types
1	Operations independent of input size (e.g., addition, min(x,y), etc.)
$\log n$	Binary search
n	Operating on every element in an array
$n \log n$	Quicksort, mergesort
n^2	Operating on every pair of items
n^3	Operating on every triple of items
2^n	Enumerating all subsets of n items
$n!$	Enumerating all orderings of n items

Python Algorithm Development Process

1. Think hard about the problem you're trying to solve. Specify the expected inputs for which you'd like to provide a solution, and the expected outputs.
2. Describe a method to solve the problem using English and/or pseudo-code
3. Start coding
   1. Development/Debugging phase
   2. Testing phase (for correctness)
   3. Evaluation phase (performance)

Let's use the insertion sort as an example of the development process in Python

Debugging in Python

1. Main strategy: run code in the interpreter to get instant feedback on errors
2. Backup: Generous use of print statements
3. Once code is running in functions: `pdb.pm()` (Python debugger post-mortem)

Main strategy: run code in the interpreter

```
>>> s = [2,7,4,5,9]
>>>
>>> for i in range(s):
...     minidx = i
...     for j in range(i,len(s)):
...         if s[j]<s[minidx]:
...             minidx=i
...             s[i],s[minidx]=s[minidx],s[i]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: range() integer end argument expected, got list.
>>> s
[2, 7, 4, 5, 9]
>>> range(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: range() integer end argument expected, got list.
>>> len(s)
5
>>> range(len(s))
[0, 1, 2, 3, 4]
```


Second strategy: print variables out during execution

```
>>> for i in range(len(s)):
...     minidx = i
...     for j in range(i,len(s)):
...         print 'list: %s, i: %i, j: %i, minidx: %i'%(s,i,j,minidx)
...         if s[j]<s[minidx]:
...             print "reassigning minidx %i < %i" %(s[j],s[minidx])
...             minidx=j
...             s[i],s[minidx]=s[minidx],s[i]
...
list: [2, 7, 4, 5, 9], i: 0, j: 0, minidx: 0
list: [2, 7, 4, 5, 9], i: 0, j: 1, minidx: 0
list: [2, 7, 4, 5, 9], i: 0, j: 2, minidx: 0
list: [2, 7, 4, 5, 9], i: 0, j: 3, minidx: 0
list: [2, 7, 4, 5, 9], i: 0, j: 4, minidx: 0
list: [2, 7, 4, 5, 9], i: 1, j: 1, minidx: 1
list: [2, 7, 4, 5, 9], i: 1, j: 2, minidx: 1
reassigning minidx 4 < 7
list: [2, 4, 7, 5, 9], i: 1, j: 3, minidx: 2
reassigning minidx 5 < 7
list: [2, 5, 7, 4, 9], i: 1, j: 4, minidx: 3
list: [2, 5, 7, 4, 9], i: 2, j: 2, minidx: 2
list: [2, 5, 7, 4, 9], i: 2, j: 3, minidx: 2
reassigning minidx 4 < 7
list: [2, 5, 4, 7, 9], i: 2, j: 4, minidx: 3
list: [2, 5, 4, 7, 9], i: 3, j: 3, minidx: 3
list: [2, 5, 4, 7, 9], i: 3, j: 4, minidx: 3
list: [2, 5, 4, 7, 9], i: 4, j: 4, minidx: 4
```

Second strategy: print variables out during execution

```
>>> for i in range(1,len(s)):
...     minidx = i
...     for j in range(i+1,len(s)):
...         print 'list: %s, i: %i, j: %i, minidx: %i'%(s,i,j,minidx)
...         if s[j]<s[minidx]:
...             print "reassigning minidx %i < %i" %(s[j],s[minidx])
...             minidx=j
...     s[i],s[minidx]=s[minidx],s[i]
...
list: [2, 7, 4, 5, 9], i: 1, j: 2, minidx: 1
reassigning minidx 4 < 7
list: [2, 7, 4, 5, 9], i: 1, j: 3, minidx: 2
list: [2, 7, 4, 5, 9], i: 1, j: 4, minidx: 2
list: [2, 4, 7, 5, 9], i: 2, j: 3, minidx: 2
reassigning minidx 5 < 7
list: [2, 4, 7, 5, 9], i: 2, j: 4, minidx: 3
list: [2, 4, 5, 7, 9], i: 3, j: 4, minidx: 3
```


Third strategy: use Python debugger

- Once you've gotten rid of the obvious bugs, move the code to a function.
- But what happens if you start getting run-time errors on different inputs?
- You can copy code directly into the interpreter
- Or you can run pdb.pm() to access variables in the environment at the time of the error

After debugging comes testing

- While you might view them as synonyms, testing is more systematic checking that algorithms work for a range of inputs, not just the ones that cause obvious bugs
- Use Python `assert` command to verify expected behavior

`assert` in action

```
>>> s
[2, 5, 4, 7, 9]
>>> t = list(s)
>>> t.sort()
>>>
>>> assert t == s
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> t
[2, 4, 5, 7, 9]
>>> s
[2, 5, 4, 7, 9]
```

Using random to generate inputs

```
>>> import random, timeit
>>> l10=range(10)
>>> l10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> random.shuffle(l10)
>>> l10
[4, 2, 0, 3, 8, 1, 9, 7, 6, 5]
>>> unsortl10 = list(l10)
>>> unsortl10
[4, 2, 0, 3, 8, 1, 9, 7, 6, 5]
>>> l10.sort()
>>> l10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> unsortl10
[4, 2, 0, 3, 8, 1, 9, 7, 6, 5]
>>> assert selection_sort(unsortl10) == l10
```


Using assert on many inputs

```
#try 10 different shufflings of each list
for i in range(10):
    #try all lists between 1 and 500 elements
    print 'trying %i time'%(i)
    for j in range(500):
        l = range(j)
        random.shuffle(l) #reorder the list
        ul = list(l)      #make a copy of the unordered list
        l.sort()          #do a known correct sort
        assert selection_sort(ul) == l  #compare sorts
```


Don't forget to look for counterexamples

- Using assert works when you have a known correct solution to compare against
- This frequently occurs when you have a known working algorithm, but you are developing a more efficient one
- While testing lots of random inputs is a good strategy, don't forget to examine edge cases and potential counterexamples too

Empirically evaluating performance

- Once you are confident that your algorithm is correct, you can evaluate its performance empirically
- Python's `timeit` package repeatedly runs code and reports average execution time
- `timeit` arguments
  1. code to be executed in string form
  2. any setup code that needs to be run before executing the code (note: setup code is only run once)
  3. parameter 'number', which indicates the number of times to run the code (default is 1000000)

Timeit in action: timing Python's sort function and our selection sort

```
#store function in file called sortfun.py
import random
def sortfun(size):
    l = range(1000)
    random.shuffle(l)
    l.sort()

>>> timeit.timeit("sortfun(1000)","from sortfun import sortfun",number=100)
0.0516510009765625
>>> #here is the wrong way to test the built-in sort function
... timeit.timeit("l.sort()","import random; l = range(1000); random.shuffle(l)"
  ,number=100)
0.0010929107666015625
>>> #let's compare it to our selection sort
>>> timeit.timeit("selection_sort(l)","from selection_sort import selection_sort;
  import random; l = range(1000); random.shuffle(l)",number=100)
3.0629560947418213
```