

Introduction to Algorithms

G. Philip

Institute of Mathematical Sciences, Chennai

October 18, 2008

What is an algorithm?

- A well-defined procedure to solve a problem.
 - Takes some input.
 - Carries out a finite number of *effective* steps.
 - Produces some output.
- An *effective* step is one which, at least in principle, can be done using pen and paper.

Problem: Pick the topper

- You have: A bunch of corrected answer books.
- You want: To find the topper's answer book.
- E.g. 38, 64, 37, 83, 65



Problem: Pick the topper

- You have: A bunch of corrected answer books.
- You want: To find the topper's answer book.
- E.g. 38, 64, 37, 83, 65



Problem: Pick the topper

- You have: A bunch of corrected answer books.
- You want: To find the topper's answer book.
- E.g. 38, 64, 37, 83, 65



Problem: Pick the topper

- Input: An array $A[1..n]$ of n numbers.
- Task: Find the (index of the) maximum number in the array.



Analysis of Algorithms

- Estimating the time taken by an algorithm.



Analysis of Algorithms

- Estimating the time taken by an algorithm.
- But wait!
 - On which input?
 - On which hardware / operating system ?
 - Written in which language?
 - ...?



Analysis of Algorithms

- Estimating the time taken by an algorithm.
 - Independent of hardware/operating system/language/... , and
 - On inputs of a *specified* size.



Analysis of Algorithms

- Estimating the time taken by an algorithm.
 - Independent of hardware/operating system/language/... , and
 - On inputs of a *specified* size.
- But wait!
 - On *which* input of size n ?



Analysis of Algorithms

- Estimating the time taken by an algorithm.
 - Independent of hardware/operating system/language/... , and
 - On inputs of a *specified* size.
- But wait!
 - On *which* input of size n ?
 - Best Case
 - Average Case
 - Worst Case



Analysis of Algorithms

- Estimating the time taken by an algorithm.
 - Independent of hardware/operating system/language/... , and
 - On *worst-case* inputs of a *specified* size, and
 - On “sufficiently large” input sizes.



The Big-Oh Notation

- A way of expressing the “most important component” of the running time, which
 - Is independent of hardware/operating system/language/... , and
 - Captures the running time behaviour on “sufficiently large” input sizes.
- Gives an *upper* bound on the growth rate of a function.



The Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, $f(n) = O(g(n))$ if there are positive constants c and n_0 such that
 - $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$
- Informally, $f(n) = O(g(n))$ if $f(n)$ is, “after some point”, not larger than a constant multiple of $g(n)$.
- Examples?



The Big-Oh Notation

- $f(n) = O(g(n))$ if there are positive constants c and n_0 such that
 - $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$
- $2n + 10 = O(n)$?
- $2n + 10 = O(n^2)$?
- $n^2 - 10 = O(n)$?
- $3n^3 + 5n^2 - 6 = O(n^2)$?
- $5\log n + \log\log n = O(\log n)$?



The Big-Oh Notation

- Gives an **upper bound** on how fast the function grows for “large enough” inputs, except for constant factors.
- “ $f(n)$ is $O(g(n))$ ” means that the growth rate of $f(n)$ **is no more than** the growth rate of $g(n)$.
- Quiz:
 - Is $2^{2n} = O(2^n)$?
 - Is $2^{n+2} = O(2^n)$?



Problem: Pick the topper

- Find the (index of the) maximum number in $A[1..n]$.

The FindMax Algorithm

```
FindMax(A[1..n])
1. maxindex := 1
2. for i := 2 to n
   do
3.   if A[i] > A[maxindex]
4.   then maxindex := i
   endif
   done
5. return maxindex
```



Problem: Pick the topper

- Find the (index of the) maximum number in $A[1..n]$.

The FindMax Algorithm

```
FindMax(A[1..n])  
1. maxindex := 1  
2. for i := 2 to n  
   do  
3.   if A[i] > A[maxindex]  
4.   then maxindex := i  
   endif  
   done  
5. return maxindex
```



Problem: Sort the answer books

- You have:
 - a bunch of corrected answer books, and
 - a way to find the topper's answer book from this bunch.
- You want: The answer books in sorted order of marks, highest marks first.
- E.g. 38, 64, 37, 83, 65



Problem: Sort the answer books

- You have:
 - a bunch of corrected answer books, and
 - a way to find the topper's answer book from this bunch.
- You want: The answer books in sorted order of marks, highest marks first.
- E.g. 38, 64, 37, 83, 65



Problem: Sort the answer books

- You have:
 - a bunch of corrected answer books, and
 - a way to find the topper's answer book from this bunch.
- You want: The answer books in sorted order of marks, highest marks first.
- E.g. 38, 64, 37, 83, 65



Problem: Sort the answer books

- Input: An array $A[1..n]$ of n numbers.
- Task: Arrange the numbers in decreasing order, using `FindMax()` as a subroutine.



Problem: Sort the answer books

- Sort the numbers in $A[1..n]$ in decreasing order.

Selection Sort

```
SelectionSort(A[1..n])
1. for i := 1 to n
   do
2.   j := FindMax(A[i..n])
3.   swap(A[i], A[j])
   done
```



Problem: Sort the answer books

- Sort the numbers in $A[1..n]$ in decreasing order.

Selection Sort

```
SelectionSort(A[1..n])
1. for i := 1 to n
   do
2.   j := FindMax(A[i..n])
3.   swap(A[i], A[j])
   done
```



Problem: Did I miss Mrs. ABC?

- You have: A large list of invitees.
- You want: To find if Mrs.ABC is in the list.



Problem: Search an Array

- Input: An array $A[1..n]$ of n numbers, and a number x .
- Task: Find whether x is present in the array.



Problem: Search an Array

- Find if a number x is present in an array $A[1..n]$ of n numbers.

Linear Search

```
LinearSearch(A[1..n],x)
1. i := 1, found := false
2. while (not found and i ≤ n)
   do
3.   if A[i] == x
4.   then found := true
   done
5. return found
```



Problem: Search an Array

- Find if a number x is present in an array $A[1..n]$ of n numbers.

Linear Search

```
LinearSearch(A[1..n],x)
1. i := 1, found := false
2. while (not found and i ≤ n)
   do
3.   if A[i] == x
4.   then found := true
   done
5. return found
```



Problem: What is the meaning of ...

- How do we look up a word in the dictionary?



Problem: Search a Sorted Array

- Input: An array $A[1..n]$ of n numbers, sorted in ascending order, and a number x .
- Task: Find whether x is present in the array.

Example:

$A = 5, 13, 16, 19, 21, 26, 34, 36, 39, 44, 48, 50, 51,$
 $54, 55, 69, 70, 71, 73, 75, 80, 82, 85, 91, 94$

$x = ?$



Problem: Search a Sorted Array

Binary Search

```
BinarySearch(A, first, last, x)
```

```
1. if (first  $\leq$  last)
```

```
    then
```

```
3.   middle =  $\lfloor (first + last) / 2 \rfloor$ 
```

```
4.   if A[middle] == x
```

```
5.   then return true
```

```
6.   else if A[middle] < x
```

```
7.   then BinarySearch(A, middle + 1, last, x)
```

```
8.   else BinarySearch(A, 1, middle - 1, x)
```

```
9. else return false
```

```
    done
```



Binary Search vs. Linear Search

- Binary Search:
 - The input should be sorted.
 - Runs in $O(\log_2 n)$ time.
- Linear Search:
 - Takes any list as input.
 - Takes $O(n)$ time.
- Is $O(\log_2 n)$ **actually** better than $O(n)$?

n	2	1024	$2^{14} = 16,384$	$2^{15} = 32,768$
$1000 \log_2 n$	1000	10,000	14,000	15,000
n	2	1024	16,384	32,768

The Divide and Conquer Method

- **Divide** the problem into a number of smaller subproblems.
- **Conquer** the subproblems by solving them recursively:
 - If a subproblem S is “small enough”, then solve it in a straightforward manner.
 - Otherwise, divide S into a number of small subproblems, ...
- **Combine** the solutions to the subproblems into the solution for the original problem.



The Divide and Conquer Method

Binary Search

BinarySearch(A, first, last, x)

1. if (first \leq last)
 then
 3. middle = $\lfloor (first + last) / 2 \rfloor$
 4. if A[middle] == x
 5. then return true
 6. else if A[middle] < x
 7. then BinarySearch(A, middle + 1, last, x)
 8. else BinarySearch(A, 1, middle - 1, x)
 9. else return false
- done



The Divide and Conquer Method

Running Time

- General form:
$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{if } n \geq d \\ c & \text{if } n < d \end{cases}$$
- $T(i)$ is the time to solve a problem instance of size i .
- n is the size of the original problem.
- a is the number of subproblems that result from a “Divide” step.
- n/b is the size of each subproblem.
- $f(n)$ is the time taken for the “Divide” and “Combine” steps.
- d is the size below which it is straightforward to solve a subproblem.
- c is a constant upper bound on the time taken to solve a “small” instance.



Solving Recurrence Relations

The Master Method

- To solve

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{if } n \geq d \\ c & \text{if } n < d \end{cases}$$

- The Master Theorem :

- 1 If $f(n) = O(n^{(\log_b a) - \epsilon})$, then $T(n) = O(n^{\log_b a})$
- 2 If $f(n) = O(n^{\log_b a})$ and $n^{\log_b a} = O(f(n))$, then $T(n) = O(n^{\log_b a} \lg n)$
- 3 If $n^{(\log_b a) + \epsilon} = O(f(n))$ and $af(n/b) \leq cf(n)$ for some $c < 1$ and all large enough n , then $T(n) = O(f(n))$.



The Master Method

Examples

- $T(n) = 4T(n/2) + n$
- $T(n) = 2T(n/2) + n \log n$
- $T(n) = T(n/3) + n \log n$
- $T(n) = 8T(n/2) + n^2$
- $T(n) = 9T(n/3) + n^3$
- $T(n) = T(n/2) + c$



Problem: Merging

- You have: Two sorted lists of numbers.
- You want: One sorted list containing all these numbers.
- E.g.

$$A = [4, 13, 16, 23, 72, 78]$$

$$B = [6, 12, 23, 38, 40, 55, 67]$$



Problem: Merging

- Merge two sorted arrays of numbers.
- Can be done in $O(n)$ time, where n is the total size of the two arrays.



Merge Sort

- Divide and Conquer: To sort an array,
 - Recursively sort two halves of the array.
 - Merge the sorted halves.
- Divide: Split the array into two nearly equal parts
- Conquer: Recursively sort each of these two parts
- Combine: Merge the sorted parts into one sorted list.



Merge Sort

Merge Sort

```
MergeSort(array,first,last)
```

1. if ($first \geq last$)
2. then return array
3. else
4. middle = $\lfloor (first + last) / 2 \rfloor$
5. ...
6. ...
7. ...
9. endif



Merge Sort

```
MergeSort(array,first,last)
```

1. if ($first \geq last$)
2. then return array
3. else
4. $middle = \lfloor (first + last) / 2 \rfloor$
5. MergeSort(array, first, middle)
6. MergeSort(array, middle + 1, last)
7. Merge(array, first, middle, last)
9. endif