Binary Search and Variants

- Applicable whenever it is possible to reduce the search space by half using one query

- Search space size N
  number of queries = O(*log* N)

- Ubiquitous in algorithm design. Almost every complex enough algorithm will have binary search somewhere.

Classic example

- Given a sorted array A of integers, find the location of a target number x (or say that it is not present)

Pseudocode:

```
Initialize start ← 0, end ← n;
Locate(x, start, end){
    if (end < start) return not found;
    mid←(start+end)/2;
    if (A[mid] = x) return mid;
    if (A[mid] < x) return Locate(x, mid+1, end);
    if (A[mid] > x) return Locate(x, start, mid-1);
}
```

Other examples

- Looking for a word in the dictionary
- Debugging a linear piece of code
- Cooking rice
- Science/Engineering: Finding the right value of any resource
  - length of youtube ads
  - pricing of a service

Egg drop problem

- In a building with n floors,
  find the highest floor from where egg can be dropped without breaking.
- $O(\log n)$ egg drops are sufficient.
- Binary search for the answer.
- Drop an egg from floor x
  - if the egg breaks, answer is less than x
  - if the egg doesn't break, the answer is at least x
- Using the standard binary search idea, start with $x=n/2$.
  If egg breaks, then go to $x=n/4$ and it doesn't then go to $x=3n/4$.
  And so on

Egg drop: unknown range

- In a building with *infinite* floors,
  find the highest floor h from where egg can be dropped without breaking.
- $O(\log h)$ egg drops sufficient?
- Exponential search: Try floors, 1, 2, 4, …, till the egg breaks.
- The egg will beak at floor 2^{k+1}, where
  $2^k \le h < 2^{k+1}$
- Then binary search in the range $[2^k, 2^{k+1}]$
- Total number of egg drops $\le k+1+\mathrm{k} = 2k+1 \le 2 \log h + 1$.
- Is there a better way?

Lower bound

- Search space size: N
  Are $\log N$ queries necessary?

- Yes. When each query is a yes/no type, then the search space gets divided into two parts with each query (some solutions correspond to yes and others to no).

- One of the parts will be at least $N/2$.

- In worst case, with each query, we get the larger of the two parts.

- To reduce the search space size to 1, we need $\log N$ queries

Lower bound

- Search space size: N
  Are $log\ N$ queries necessary?
- Another argument based on information theory.
- Each yes/no query gives us 1 bit of information.
- The final answer is a number between 1 and N, and thus, requires $log\ N$ bits of information.
- Hence, $log\ N$ queries are necessary.
- Ignore this argument if it is hard to digest.

Exercise: subarray sum

- Given an array with n positive integers,
  and a number S,
  find the minimum length subarray whose sum is at least S?

- Subarray is a contiguous subset, i.e.,
  A[i], A[i+1], A[i+2], …, A[j-1], A[j]

- [10, 12, 4, 9, 3, 7, 14, 8, 2, 11, 6]

  S = 27

- Can we do this in *O(n log n) time*?

Exercise: subarray sum

O(n^2) algorithm:

for ($l \leftarrow$ 1 to n){

T $\leftarrow$ sum of first l numbers.

for ($j \leftarrow 0$ to n-l-1){

if T $\geq$ S return l and the subarray (j, j+l-1);

T $\leftarrow$ T - $A[j]$ + $A[l+j]$;

}

}

- Two for loops one inside the other. Each makes at most n iterations. Hence, O(n^2) time.

Exercise: subarray sum

O($n \log n$) algorithm. Approach 1:

Binary search for the minimum length l.

For the current value of l:

check if there is a subarray of length l with sum at least S.
This check can be done in $O(n)$ time. See the inner loop on previous slide.

Exercise: subarray sum

- O($n \log n$) algorithm. Approach 2 (suggested by students):
- First compute all the prefix sums and store in an array.
  $O(n)$ time.
- prefix_sum[0] ← A[0];
  for (i ← 1 to n-1)
  prefix_sum[i] = prefix_sum[i-1]+A[i];
- Any subarray sum from i to j can now be computed in O(1) time as prefix_sum[j] - prefix_sum[i-1].
- Now, for each choice of starting point, do a binary search for the minimum end point such that the subarray sum is at least S.
- If sum of a subarray < S, then choose a larger end point, otherwise smaller.

Exercises

- Given two sorted arrays of size n, find the median of the union of the two arrays.
  $O(\log n)$ time?

- Given a convex function $f(x)$ oracle, find an integer x which minimizes $f(x)$.

- Land redistribution:
  given list of landholdings $a_1, a_2, \ldots, a_n$,
  given a floor value f,
  find the right ceiling value c

Division algorithm

- As we find the next digit of the quotient,
  the search space of the quotient goes down by a factor of 10.

- This could be called a denary search.

- For binary representation of numbers, the division algorithm will be a binary search.

Finding square root

- Given an integer a, find $\sqrt{a}$
- Start with a guess x
- If $x^2 > a$, then the answer is less than x
- Else the answer is at least x.
- This way we can do binary search for $\sqrt{a}$
- Additional exercise: find a division like algorithm.

Finding square root

- Given an integer a, find $\sqrt{a}$ up to l digits after decimal.
- Can we compute it in $O(l)$ iterations?
- Yes.

More applications

- Finding inverse of an increasing function?

- Finding root of polynomial?

- Finding the smallest prime dividing N ?

- Is sorting a kind of binary search?
  $O(n \log n)$ comparisons necessary?

More applications

- Finding inverse of an increasing function?
- For any given x, we have a method to compute $f(x)$. For a given y, we want to compute $f^{-1}(y)$.
- Make a guess x and compare it $f(x)$ with y
- If $f(x) < y$ then the answer is larger than x
- Else the answer is at least x.

More applications

- Finding a root of polynomial $f(x)$?
- Always maintain two points a and b such that $f(a) > 0$ and $f(b) < 0$.
- To find the starting points one can do exponential search.
- Check whether $f((a+b)/2) > 0$.
- If yes, then there is a root between $(a+b)/2$ and b.
- Else, there is a root between $(a+b)/2$ and a.

More applications

- Finding the smallest prime dividing N ?
- No.
- We can make a guess x. If x does not divide N, then we cannot say anything about where should be the smallest prime dividing N.

More applications

- Is sorting a kind of binary search?
  $n \log n$ comparisons necessary?
- Yes.
- When we have not made any comparisons, then any of the $n!$ rearrangements is a possible answer.
- So the search space size is $n!$.
- Each comparison will reduce the search space size by only 1/2 (in worst case).
- Hence, $\log (n!) \geq n \log n - n \log e$ comparisons are necessary.

Analyzing algorithms

- Comparison between various candidate algorithms
- Why not implement and test?
  - too many algorithms
  - depends on input size, how inputs are chosen
- Will count the number of basic operations like addition, comparison etc.
- And see how this number grows as a function of the input size. This measure is independent of the choice of the machine.

$O(\cdot)$ notation

- For input size n, running time $f(n)$

- We say $f(n)$ is $O(T(n))$
  if for all large enough n and some constant c,
  $f(n) \leq c \cdot T(n)$

$O(\cdot)$ notation

- Why do we ignore constant factors?

- Because it's not possible to find the precise constant factor. Various basic operations do not take the same amount of time.

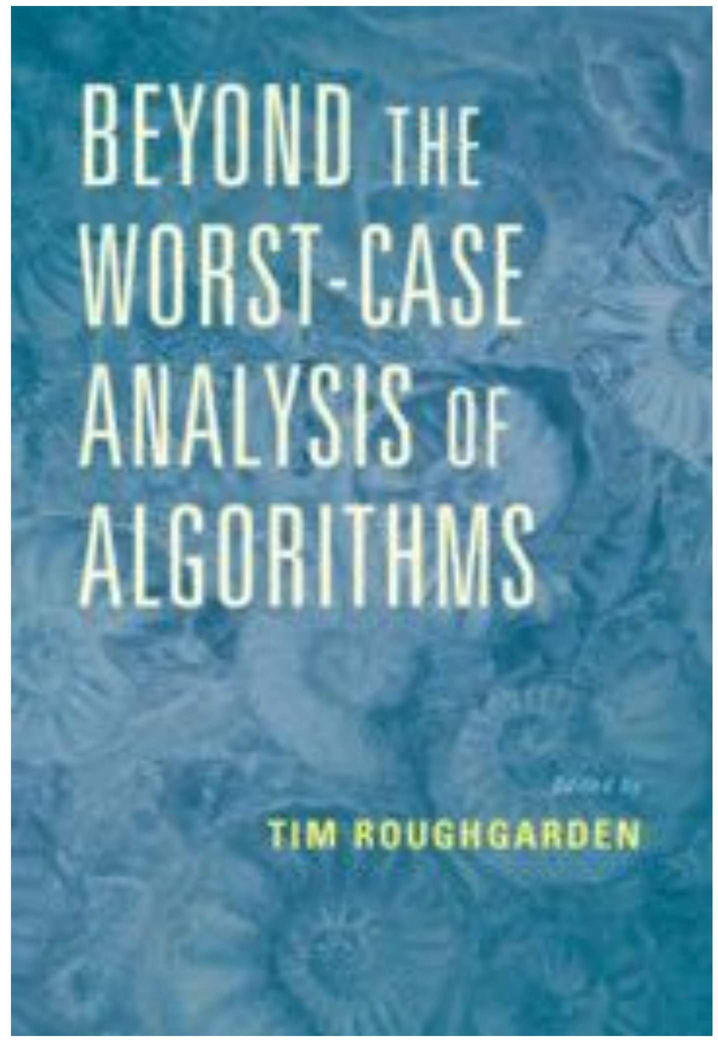
- Is $O(n)$ always better than $O(n \log n)$?

- For large enough inputs, yes. But, depending on the hidden constant factors, it's possible that $O(n \log n)$ algorithm is faster on reasonable size inputs.

Worst case analysis

- Worst case bound: running time guarantee for all possible inputs of a size.
- There could be algorithms which are slow on a few pathological instances, but otherwise quite fast.
- Why not analyze only for "real world inputs"?
- It's not clear how to model "real world inputs".
- For many algorithms, we are able to give worst case bounds.

Worst case analysis



Out of scope of this course

Describing algorithms

- Find the maximum sum subarray of length k

$s = 0$;
for (i=0, $i < k$, i++) s=s+$A[i]$;
$m = s$;
for (i=0, $i < n$-k, i++){
    $s = s - A[i] + A[k+i]$;
    if (s > m) m = s;
}
return m;
	Compute the sum of first k numbers. We will go over all length k subarrays from left to right. In an iteration, update the sum by subtracting the first number of the current array and adding the number following the last one. If the sum is larger than the maximum seen so far, we update the maximum.	s $\leftarrow$ sum of first k numbers;
		for ($i \leftarrow 0$ to n-k-1){
		    $s \leftarrow s - A[i] + A[k+i]$;
		    $m \leftarrow \max(m, s)$;
		}
		return m;

Precise, but hard to understand. Error prone.	Not precise. Open to multiple interpretations.	Somewhere in the middle.

Describing algorithms

- Two sorted (increasing) arrays A and B of length n. Count pairs (a,b) such that $a \in A$ and $b \in B$ and $a < b$

```
j=0; count = 0;
for (i=0, i < n, i++){
    while (A[i] >= B[j]) j++;
    count=count + n - j;
}
return count;
```

```
j ← 0; count ← 0;
for (i ← 0 to n-1){
    keep increasing j till we get B[j] > A[i];
    count ← count + n - j;
}
return count;
```

O(·) notation

- True or False?

- $2n+3$ is $O(n^2)$
  - True
- $1^2 + 2^2 + \ldots. + (n-1)^2 + n^2$ is $O(n^2)$
  - False (it is $\Theta(n^3)$)
- $1 + 1/2 + 1/3 + \ldots + 1/n$ is $O(\log n)$
  - True
- n^n is $O(2^n)$
  - False

$O(\cdot)$ notation

- True or False?
- 2^{3n} is $O(2^n)$
  - False
- $(n+1)^3$ is $O(n^3)$
  - True
- $(n+\sqrt{n})^2$ is $O(n^2)$
  - True
- $\log(n^3)$ is $O(\log n)$
  - True

Principles of algorithm design

First principle: reducing to a subproblem

- Subproblem: same problem on a smaller input
- Assume that you have already built the solution for the subproblem and using that try to build the solution for the original problem.
- Subproblem will be solved using the same strategy.
- Implementation: recursive or iterative

First principle: reducing to a subproblem

- Example 1: finding minimum value in an array.
- Suppose we have already found minimum among first $n\text{-}1$ numbers, say min_{n-1}
- $min_n = \mathrm{minimum}(\ min_{n-1},\ A[n]\ )$
- Iterative implementation:
  Go over the array from 1 to n and maintain a variable min
- Invariant: after seeing i numbers, min will be the minimum among first i numbers.
- $min_i = \mathrm{minimum}(\ min_{i-1},\ A[i]\ )$

First principle: reducing to a subproblem

- min_i = minimum(min_{i-1}, $A[i]$)

- Iterative implementation

$min \leftarrow A[1]$;

for ($i \leftarrow 2$ to n)

  $min \leftarrow$ minimum(min, $A[i]$)

- Recursive implementation

$f(A, i)$:

  if $i=1$ return $A[1]$;

  else return minimum($f(A, i-1)$, $A[i]$);

Compute $f(A, n)$;

Maximum subarray sum

- Given an integer array with positive/negative numbers. Find the subarray with maximum possible sum.
- 1, 2, -5, 4, -6, $\boxed{8, 7, -3, 2, 10, 3,}$ -7, 4, 2
- $O(n^2)$ algorithm
- For every choice of starting point, go over all choices of end points and maintain the sum between starting and end points.
- Maintain the *max_sum* value by comparing with the current sum

Maximum subarray sum

$O(n^2)$ algorithm:

```
max_sum ← 0;
for (start ← 1 to n){
        curr_sum ← 0;
        for (end ← start to n){
            curr_sum ← curr_sum + A[end];  // update the current sum
            max_sum ← maximum(curr_sum, max_sum);
        }
}
```

Max subarray sum: subproblem

- Suppose we have already found the maximum subarray sum for $A[1 \ldots n\text{-}1]$, say max_sum_{n-1}
- How do we compute max_sum_n
- Two kinds of subarrays of A:
  1. subarrays of $A[1 \ldots n\text{-}1]$
  2. subarrays of A which end at n
- $max\_sum_n =$ Maximum(max_sum_{n-1} ,

$$\left.\begin{array}{l} Sum(1 \ldots n), \\ Sum(2 \ldots n), \\ \quad\vdots \\ Sum(n \ldots n), \;) \end{array}\right\} O(n)$$

$$T(n) = T(n\text{-}1) + O(n) \qquad \Rightarrow T(n) = O(n^2)$$

Max subarray sum: subproblem

- Improvement ?

- Observation:

$$Sum(1 \ldots n) = Sum(1 \ldots n\text{-}1) + A[n],$$
$$Sum(2 \ldots n) = Sum(2 \ldots n\text{-}1) + A[n],$$
$$\vdots$$
$$Sum(n\text{-}1 \ldots n) = Sum(n\text{-}1 \ldots n\text{-}1) + A[n],$$

- Maximum($Sum(1 \ldots n), Sum(2 \ldots n), \ldots, Sum(n\text{-}1 \ldots n)$)

  =

  Maximum($Sum(1 \ldots n\text{-}1), Sum(2 \ldots n\text{-}1), \ldots, Sum(n\text{-}1 \ldots n\text{-}1)$) + $A[n]$

- We have converted it to another problem on first n-1 numbers

Max subarray sum: subproblem

- Improvement ?
- Observation:

$$Sum(1 \ldots n) = Sum(1 \ldots n\text{-}1) + A[n],$$
$$Sum(2 \ldots n) = Sum(2 \ldots n\text{-}1) + A[n],$$
$$\vdots$$
$$Sum(n\text{-}1 \ldots n) = Sum(n\text{-}1 \ldots n\text{-}1) + A[n],$$

Push it to the subproblem

- Maximum($Sum(1 \ldots n), Sum(2 \ldots n), \ldots, Sum(n\text{-}1 \ldots n)$)

=

Maximum($Sum(1 \ldots n\text{-}1), Sum(2 \ldots n\text{-}1), \ldots, Sum(n\text{-}1 \ldots n\text{-}1)$) + $A[n]$

- We have converted it to another problem on first $n\text{-}1$ numbers

Asking subproblem to do more

- Subproblem: max_sum_{n-1} and $max\_suffix\_sum_{n-1}$

- $max\_suffix\_sum_{n-1}$ is defined as
  Maximum($Sum(1 \ldots n\text{-}1), Sum(2 \ldots n\text{-}1), \ldots Sum(n\text{-}1 \ldots n\text{-}1)$)

- $max\_sum_n =$ Maximum(max_sum_{n-1} ,
  $Sum(1 \ldots n\text{-}1) + A[n],$
  $Sum(2 \ldots n\text{-}1) + A[n],$
  $\vdots$
  $Sum(n\text{-}1 \ldots n\text{-}1) + A[n],$
  $A[n]$)

- $=$ Maximum(max_sum_{n-1} ,
  $max\_suffix\_sum_{n-1} + A[n],$
  $A[n]$)

Asking subproblem to do more

- Subproblem: max_sum_{n-1} and $max\_suffix\_sum_{n-1}$
- $max\_sum_n =$ Maximum(max_sum_{n-1} ,
  $max\_suffix\_sum_{n-1} + A[n]$,
  $A[n]$)
- We are asking the subproblem to compute max_suffix_sum for size n-1
  So, we also need to compute max_suffix_sum for size n
- $max\_suffix\_sum_n =$?
- $max\_suffix\_sum_n =$ Maximum($max\_suffix\_sum_{n-1} + A[n]$,
  $A[n]$)

$T(n) = T(n\text{-}1) + O(1) \quad \Rightarrow T(n) = O(n)$

Maximum subarray sum

O(n) algorithm:

$max\_sum \leftarrow 0$; $max\_suffix\_sum \leftarrow 0$;

for ($i \leftarrow 1$ to n){

$max\_sum \leftarrow \text{maximum}(max\_sum, max\_suffix\_sum + A[i], A[i])$;

$max\_suffix\_sum \leftarrow \text{maximum}(max\_suffix\_sum + A[i], A[i])$;

}

Alternate implementation

$max\_sum \leftarrow 0$; $max\_suffix\_sum \leftarrow 0$;

for ($i \leftarrow 1$ to n){

$max\_suffix\_sum \leftarrow \text{maximum}(A[i], max\_suffix\_sum + A[i])$;

$max\_sum \leftarrow \text{maximum}(max\_suffix\_sum, max\_sum)$;

}

- Here we are updating the two variables in a different order.

Reducing to a subproblem

- When solving a problem recursively/inductively, it is sometimes useful to solve a more general problem

- Stronger induction hypothesis

Exercises

- Given share prices for *n* days
  $p_1, p_2, \ldots, p_n$

- You have to buy it on one of the days and sell it on a later day.

- Maximum profit possible in $O(n)$?

- $\max_{\{j > i\}} (p_j - p_i)$

Exercises

- There is a party with n people, among them there is 1 celebrity.
- A celebrity is someone who is known to everyone, but she does not know anyone.
- you ask the any person i if they know person j.
- Can you do find the celebrity in $O(n)$ queries?