CS5800 – ALGORITHMS

MODULE 1. REVIEW OF ASYMPTOTIC NOTATION

Lesson 1: Search

Ravi Sundaram

Topics

- Algorithms
- Search – unordered array
  - Linear search
- Search – ordered array
  - Linear search
  - Chunk search
  - Binary search
- Cellphone drop puzzle
- Summary

Algorithms

- An algorithm is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation
- Important to specify the form of the input, the output and model of computer
- We care about resource usage
  - Time complexity
  - Space complexity
- What is the metric for resource usage?

Search – unordered array

- Find element x in an array of n distinct elements A[1], A[2], … A[n]. (i.e. find i such that x = A[i], else return not-found)
- LinearSearch

```
For i = 1 To n
    If x = A[i] Then Return(i)
Endfor
Return(not-found)
```

Unordered search – time complexity

- Upper bound on LinearSearch
  - Could get lucky and find a match on the first array element
  - Given an array of n numbers, makes at most n comparisons

- We parameterize the time complexity in terms of the input size, n
- We assume that comparisons take one unit of time in our model of computation
- We consider the worst case as it is a guaranteed upper bound.
- Time complexity is $O(n)$
- Lower bound on the problem
  - Only slightly less obvious, *any* comparison-based algorithm must compare x to every element in the array, otherwise if it did not compare to some element then we could change the value of the element to invalidate the correctness of the algorithm.
  - Hence any comparison-based algorithm must make at least n comparisons.

Ordered array – Linear Search

- Find element x in an *ordered* array of n distinct elements A[1] < A[2] < … < A[n] . (i.e. find i such that x = A[i] else return not-found)
- Linear search

```
For i = 1 To n
    If x = A[i] Then Return(i)
Endfor
Return(not-found)
```

- Time complexity continues to be O(n)
- Can we exploit the order structure in the input to gain performance, i.e. reduce time complexity?

Ordered array – Chunk Search

- Think of array as broken into contiguous “chunks” of size c, i.e.

1st chunk | 2nd chunk | (n/c)th chunk

A[1], A[2] … A[c], A[c+1], A[c+2], … A[2c], … A[n-c+1], A[n-c+2], … A[n]

- Do a linear search over the subsequence comprised of the last elements of each chunk to find a potential chunk where x might be located and then do linear search over that chunk
- Idea: if x is greater than the last element of a chunk then it could not possibly be located in that chunk and we can skip the chunk entirely saving a chunk-worth of comparisons

Ordered array – Chunk Search

- ChunkSearch

```
For i = c To n, step c
    If x ≤ a_i Then For j = i-c+1 to i
                        If x = a_j Then Return(j)
                    Endfor
                    Return(not-found)
Endfor
Return(not-found)
```

Ordered array – Chunk Search

- What is the time complexity of Chunk-Search?
- Again, we consider the worst-case. In the worst case we compare with the entire subsequence of last elements and then compare against the entire last chunk.
- #comparisons = n/c + c
- Remember, we control c and can choose c to minimize the above. Balance #chunks & chunk-size

  $n/c = c$ or $c = \sqrt{n}$ => #comparisons = $2\sqrt{n}$
- Time complexity is $O(\sqrt{n})$
- Can we exploit the order structure yet more?

Ordered array – Binary Search

- Instead of linear search over the subsequence of last elements we could break them into chunks again or 2-level chunk search! What is best chunk hierarchy?
- Key idea: when we compare x with A[i] then depending on the outcome we can ignore all elements before or after i
- BinarySearch(low, high)

```
If (high < low) Then Return(not-found)
mid = (low + high) / 2
If (A[mid] > x) Then Return(BinarySearch(low, mid-1))
ElseIf (A[mid] < x) Then Return(BinarySearch(mid+1, high))
Else Return(mid)
```

Ordered array – Binary Search

- What is the time complexity of BinarySearch?
- Recursive algorithm. Recurrence: $T(n) = T(n/2) + 1$; $T(1)=1$
- Solution: $T(n) = T(n/2) + 1$

  $= T(n/2^2) + 2$

  $= \ldots$

  $= T(n/2^k) + k$

  $= T(n/2^{\lg n}) + \lg n$

  $= O(\lg n)$
- Time complexity is $O(\lg n)$. And this is tight.

Cellphone-Drop puzzle

- You are a summer intern at Samsung hired to test the breakability of cellphones. Your manager gives you one cellphone and asks you to find the lowest floor from which it will break when dropped from that floor. You know that it doesn't break when dropped from the ground floor and it definitely breaks when dropped from the roof. What is the minimum number of drops you need to try in the worst case?
- What if you had 2 cellphones?
- 3 cellphones?
- An unlimited number of cellphones?

Summary

- An algorithm is a recipe that a computer uses to convert its input into the desired output.
- We care about time and space complexity and parameterize them in terms of n the size of the input.
- We work with an abstract model of the computer.
- Main Takeaway: the complexity of an algorithm is a worst-case measure, parameterized in terms of input size