No HW is due in midterm week.

Introduction to Algorithms

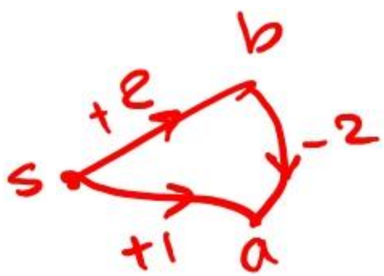
Dijkstra’s Algorithm,
Divide and Conquer

Remarks on Dijkstra's Algorithm

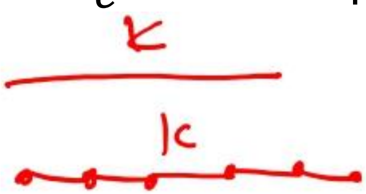
- Algorithm also produces a tree of shortest paths to s following Parent links
- Algorithm works on directed graph (with nonnegative weights)

- The algorithm fails with negative edge weights.
  - e.g., some airline tickets

Why does it fail?



- Dijkstra's algorithm is similar to BFS:
  - Subtitute every edge with $c_e = k$ with a path of length k, then run BFS.



dijk = BFS.

Implementing Dijkstra's Algorithm

Priority Queue: Elements each with an associated key Operations

- Insert
- Find-min
  - Return the element with the smallest key
- Delete-min
  - Return the element with the smallest key and delete it from the data structure
- Decrease-key
  - Decrease the key value of some element

Implementations

Arrays:

- $O(n)$ time find/delete-min,
- $O(1)$ time insert/decrease key

Binary Heaps:

- $O(\log n)$ time insert/decrease-key/delete-min,
- $O(1)$ time find-min

Dijkstra’s Algorithm

Runs in $O(|E|+|V|^2)$.

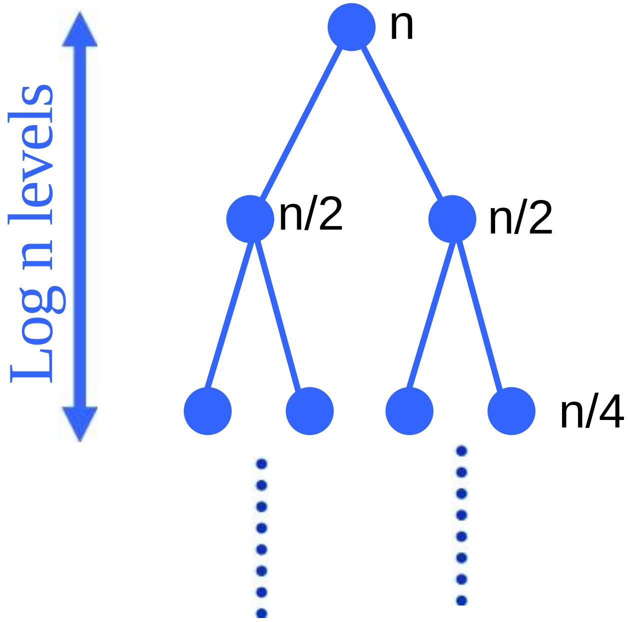
```
function Dijkstra(Graph, source):
    create vertex set Q
    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0
    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q
        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
    return dist[], prev[]
```

Divide and Conquer Approach

Divide and Conquer

Similar to algorithm design by induction, we reduce a problem to several subproblems.

Typically, each sub-problem is at most a constant fraction of the size of the original problem



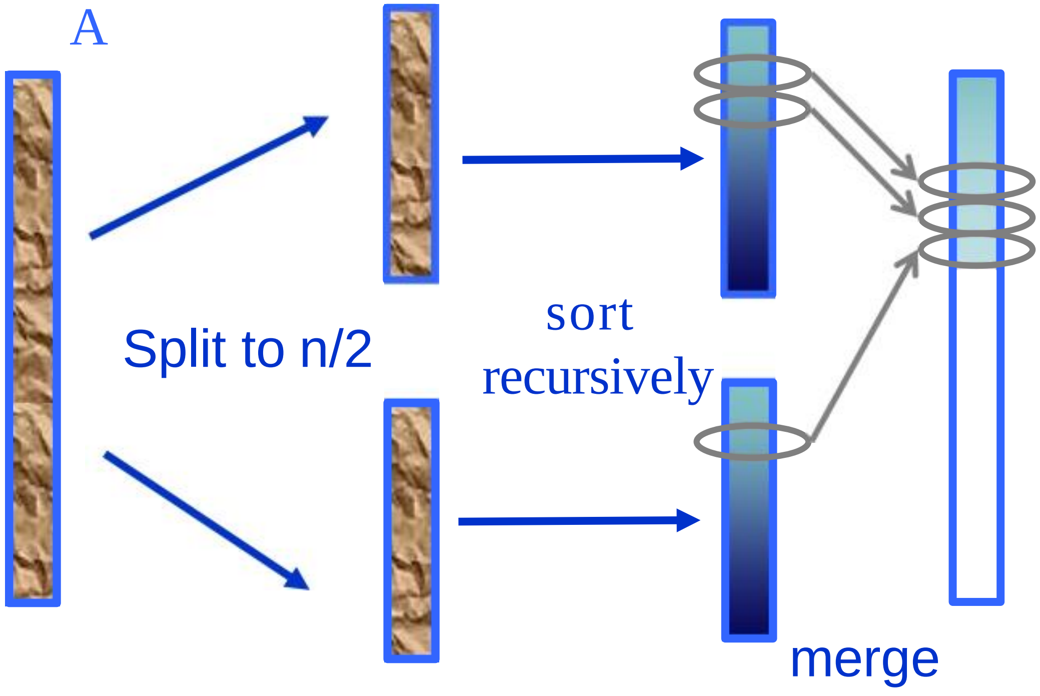
Recursively solve each subproblem

Merge the solutions

Examples:

- Mergesort, Binary Search, Strassen’s Algorithm,

A Classical Example: Merge Sort



Why Balanced Partitioning?

An alternative "divide & conquer" algorithm:

- Split into n-1 and 1
- Sort each sub problem
- Merge them

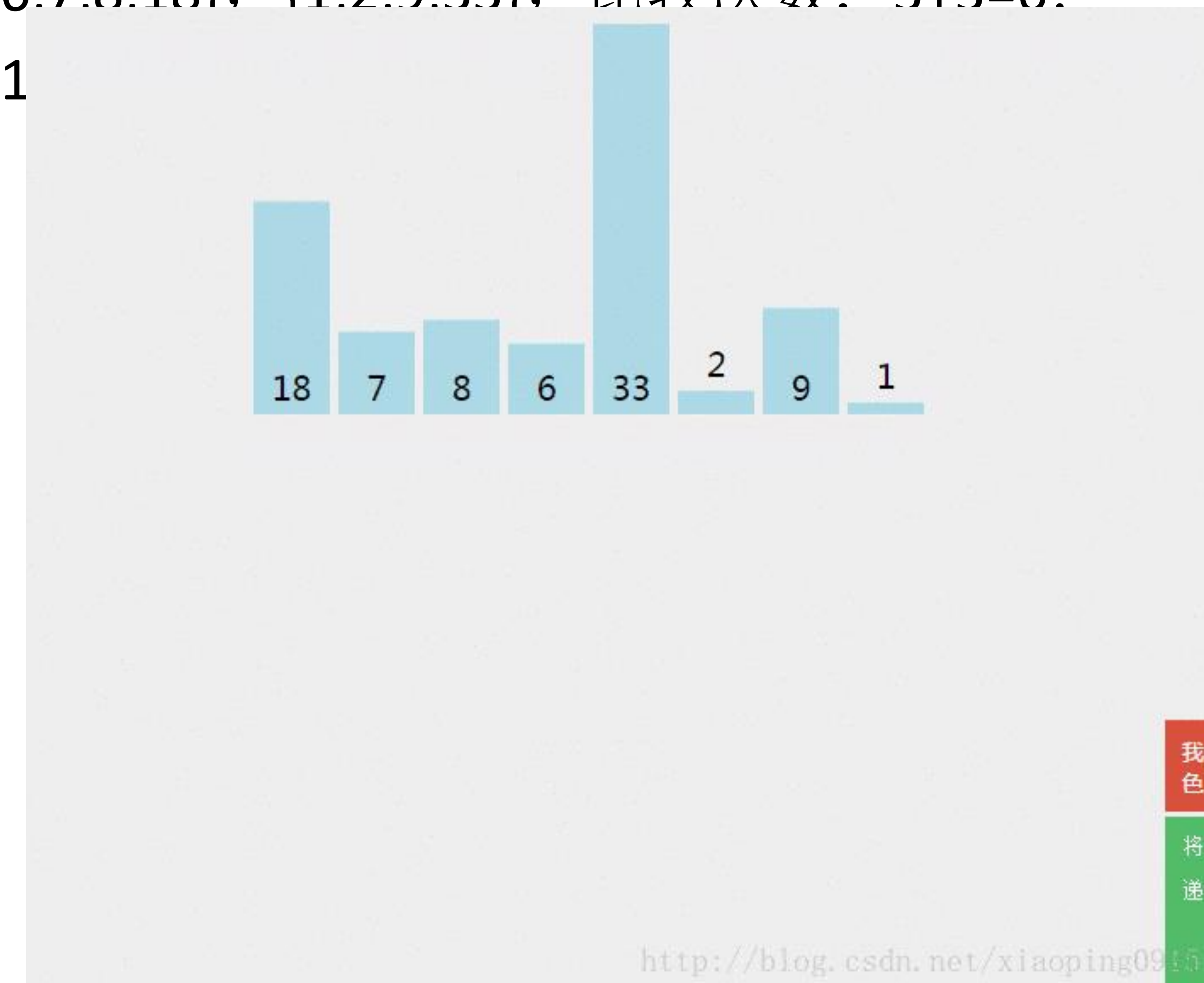
Runtime

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

Solution:

$$\begin{aligned} T(n) &= n + T(n\text{-}1) + T(1) \\ &= n + n - 1 + T(n\text{-}2) \\ &= n + n - 1 + n - 2 + T(n\text{-}3) \\ &= n + n - 1 + n - 2 + \cdots + 1 = O(\mathrm{n}^2) \end{aligned}$$

- 如设有数列{18, 7, 8, 6, 33, 2, 9, 1}
- 初始状态： 18, 7, 8, 6, 33, 2, 9, 1
- 第一次归并后：{7,18}, {6,8}, {2,33}, {1,9}，比较次数：4；
- 第二次归并后：{6,7,8,18}，{1,2,9,33}，比较次数：3+3=6；
- 第三次归并后：{1
- 总的比较次数为：



D&C: The Key Idea

Suppose we've already invented Bubble-Sort, and we know it takes n^2

Try just one level of divide & conquer:

Bubble-Sort(first n/2 elements)

Bubble-Sort(last n/2 elements)

Merge results

Time: $2\underbrace{T(n/2)}_{n^2/4} + n = n^2/2 + \mathrm{n} \ll n^2$

Almost twice as fast!

D&C in a nutshell

D&C approach

- “the more dividing and conquering, the better”
  - Two levels of D&C would be almost 4 times faster, 3 levels almost 8, etc., even though overhead is growing.
  - Best is usually full recursion down to a small constant size (balancing "work" vs "overhead").

  In the limit: you’ve just rediscovered mergesort!
- Even unbalanced partitioning is good, but less good
  - Bubble-sort improved with a 0.1/0.9 split:

    $$(.1n)^2 + (.9n)^2 + n = .82n^2 + 1$$

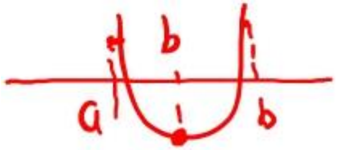
    The 18% savings compounds significantly if you carry recursion to more levels, actually giving $O(n \log n)$, but with a bigger constant.
- This is why Quicksort with random splitter is good – badly unbalanced splits are rare, and not instantly fatal.

Finding the Root of a Function

Finding the Root of a Function

Given a continuous function f and two points a < b such that

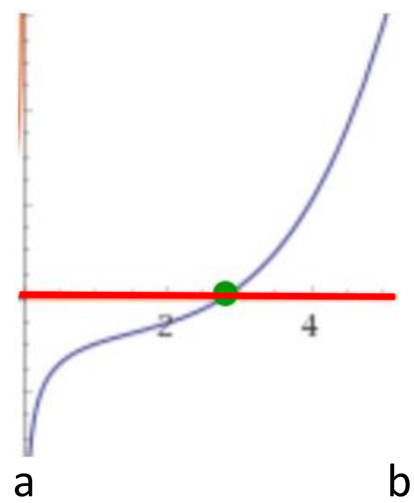
$$f(a) \le 0$$
$$f(b) \ge 0$$



Find an approximate root of *f* (a point c where $f(c) = 0$).

f has a root in $[a, b]$ by intermediate value theorem

$$f(x) = \sin(x) - \frac{100}{\sqrt{x}} + x^4$$



Note that roots of f may be irrational,
So, we want to approximate
the root with an arbitrary precision!

A Naiive Approch

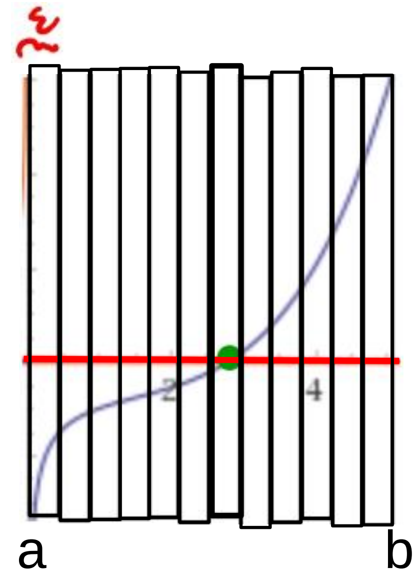
Suppose we want ϵ approximation to a root.

Divide [a,b] into $n = \frac{b-a}{\epsilon}$ intervals. For each interval check

$$f(x) \leq 0, f(x+\epsilon) \geq 0$$

This runs in time $O(n) = O(\frac{b-a}{\epsilon})$

Can we do faster?



D&C Approach (Based on Binary Search)

Bisection(a,b, ε)

if $(b - a) < \epsilon$ then

return (a)

else

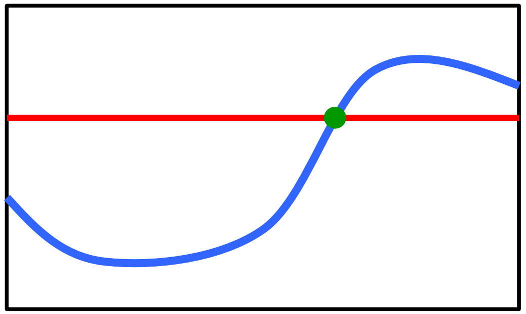
$c \leftarrow (a + b)/2$

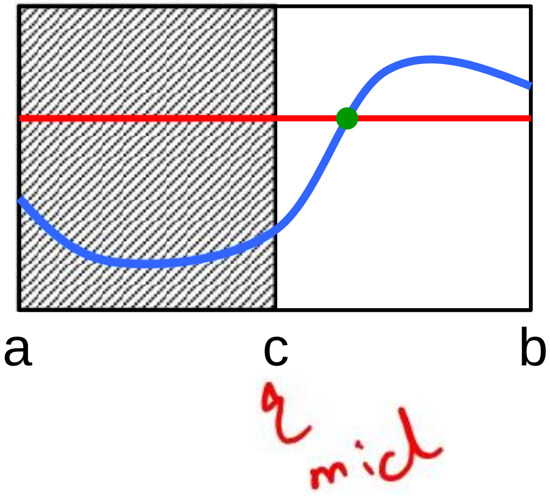
if $f(c) \leq 0$ then

return(Bisection(c, b, ε))

else

return(Bisection(a, c, ε))



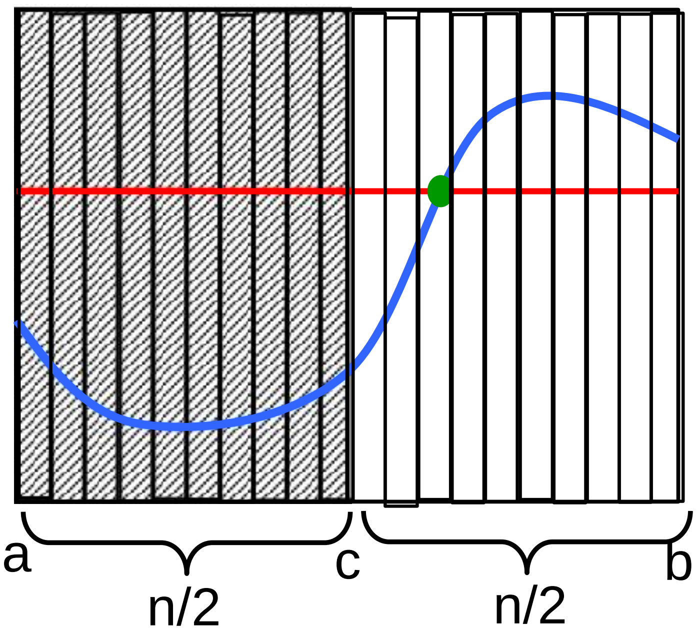


Time Analysis

Let n = $\frac{a-b}{\epsilon}$

And $c = (a+b)/2$

Always half of the intervals lie to the left and half lie to the right of c



So,

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

i.e., $T(n) = O(\log n) = O\left(\log \frac{a-b}{\epsilon}\right)$