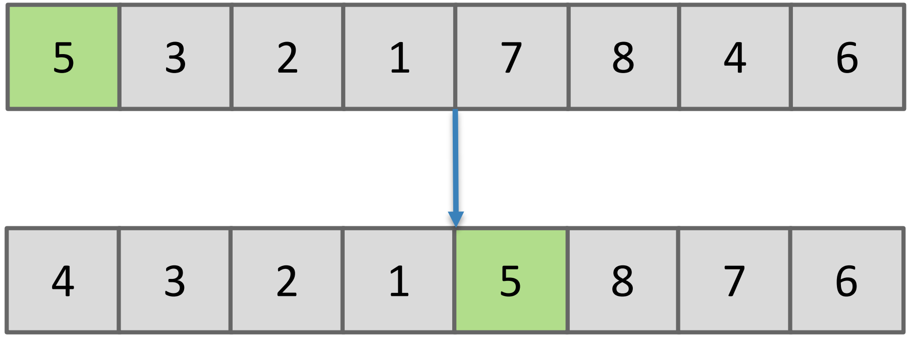
Discussion 12

— Median of Medians & Quick Sort

Quicksort

Reference to Berkeley CS61b, https://docs.google.com/presentation/d/19CbZ7oU-_b5y0Btyo1I

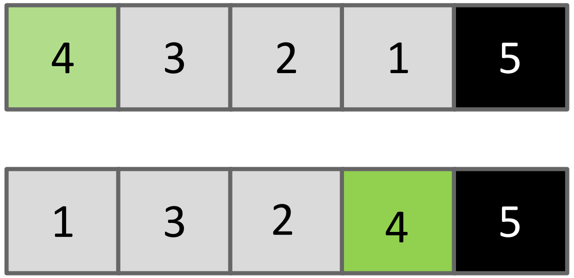
Partition Sort, a.k.a. Quicksort

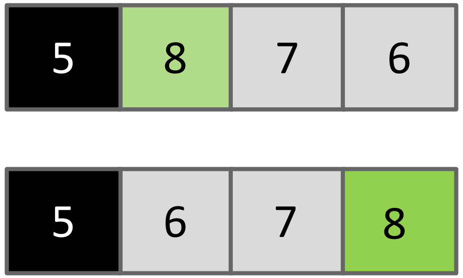


Q: How would we use this operation for sorting?

Observations:

- 5 is "in its place." Exactly where it'd be if the array were sorted.
- Can sort two halves separately, e.g. through recursive use of partitioning.





Quicksort

Quicksort was the name chosen by Tony Hoare for partition sort.

- For most common situations, it is empirically the fastest sort.
    - Tony was lucky that the name was correct.

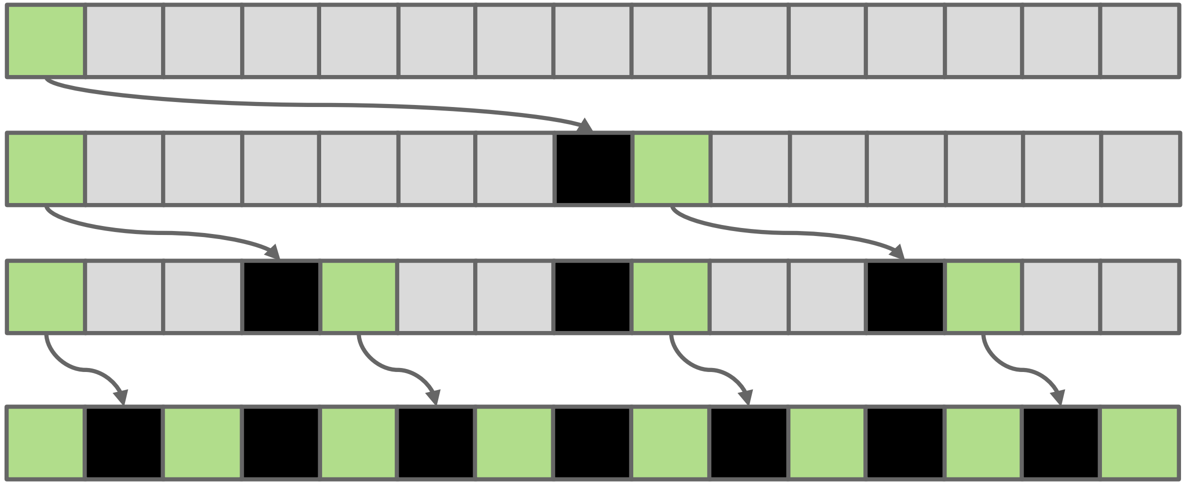
How fast is Quicksort? Need to count number and difficulty of partition operations.

Theoretical analysis:

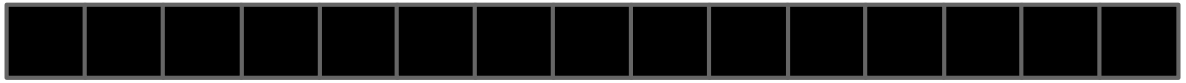
- Partitioning costs Θ(K) time, where Θ(K) is the number of elements being partitioned (as we saw in our earlier "interview question").
- The interesting twist: Overall runtime will depend crucially on where pivot ends up.

Quicksort Runtime

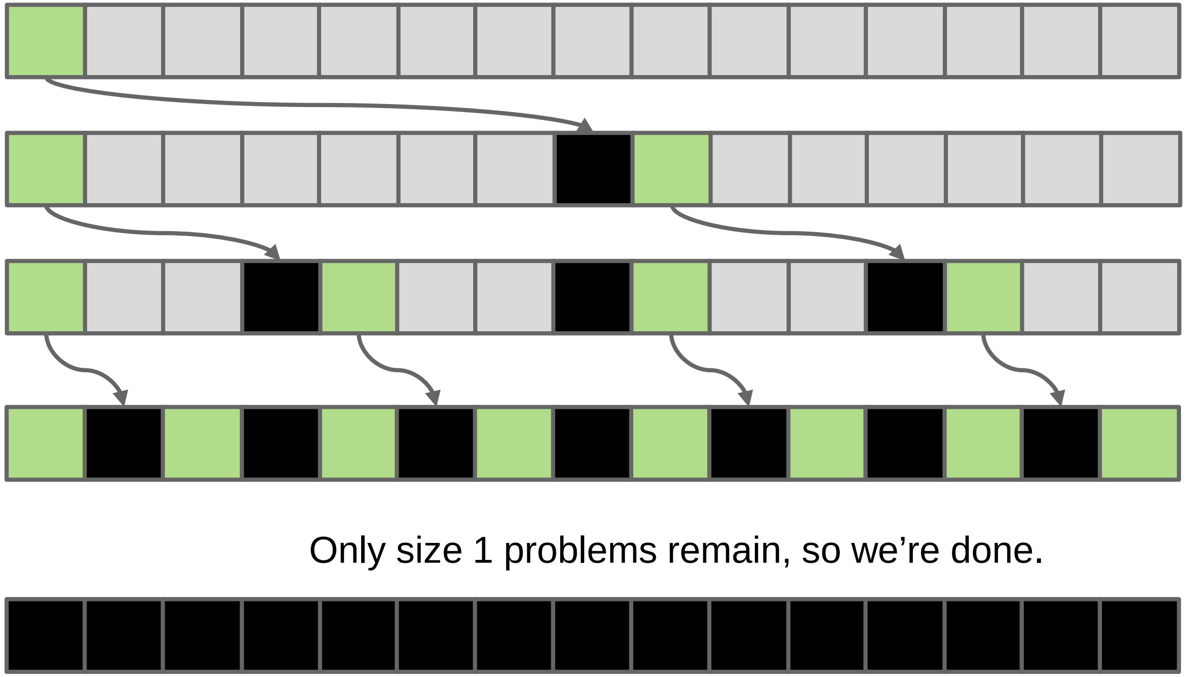
Best Case: Pivot Always Lands in the Middle



Only size 1 problems remain, so we're done.

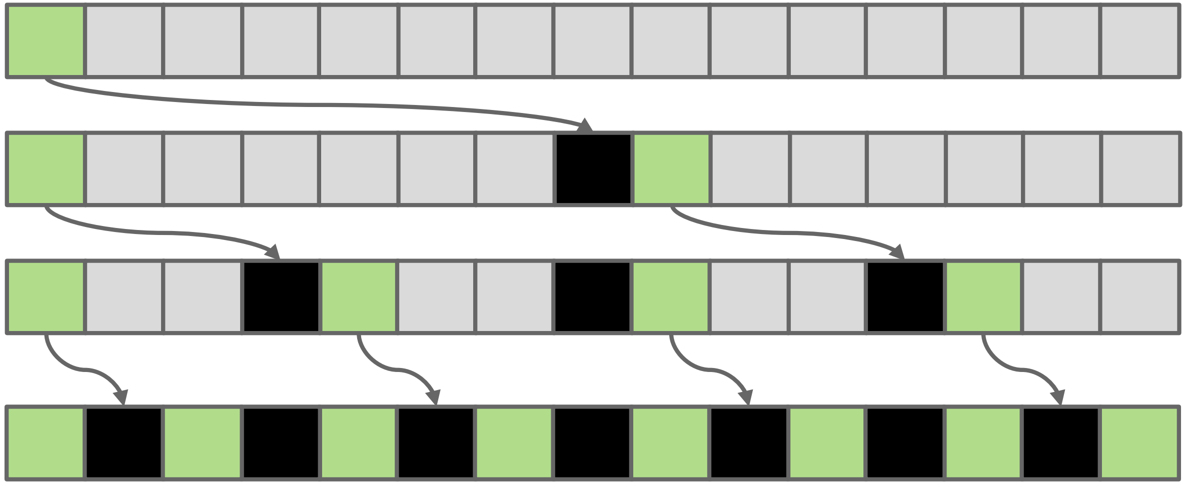


Best Case Runtime?


Only size 1 problems remain, so we're done.

What is the best case runtime?

Best Case Runtime?



Only size 1 problems remain, so we're done.

Total work at each level:

≈ N

≈N/2 + ≈N/2 = ≈N

≈N/4 * 4 = ≈N

Overall runtime:

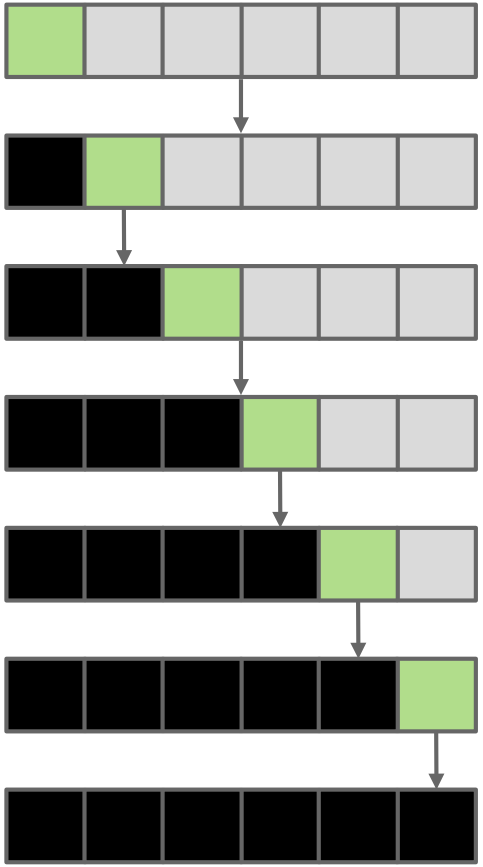
Θ(NH) where H = Θ(log N)

so: Θ(N log N)

Worst Case: Pivot Always Lands at Beginning of Array

Give an example of an array that would follow the pattern to the right.

What is the runtime Θ(·)?



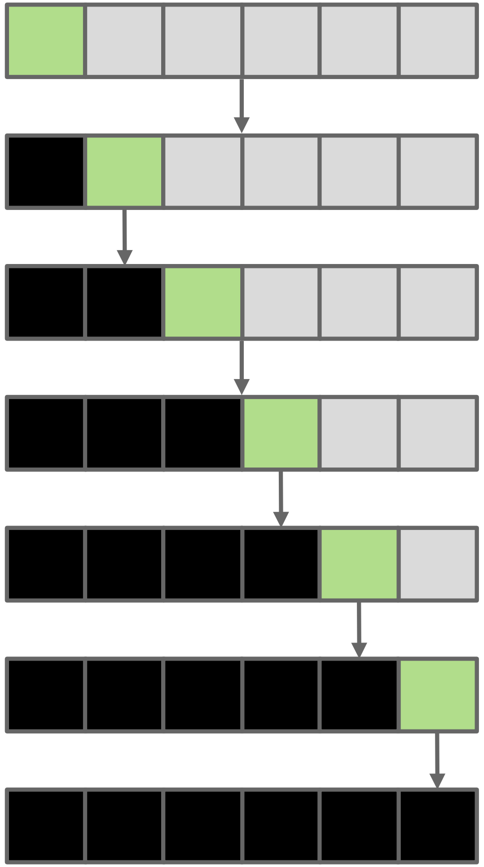
Worst Case: Pivot Always Lands at Beginning of Array

Give an example of an array that would follow the pattern to the right.

- 1 2 3 4 5 6

What is the runtime Θ(·)?

- N^2



Quicksort Performance

Theoretical analysis:

- Best case: $\Theta(N \log N)$
- Worst case: $\Theta(N^2)$

Compare this to Mergesort.

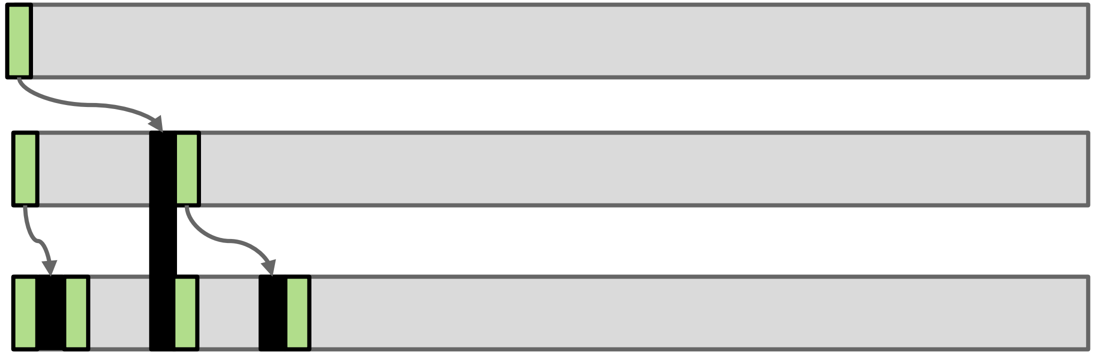
- Best case: $\Theta(N \log N)$
- Worst case: $\Theta(N \log N)$

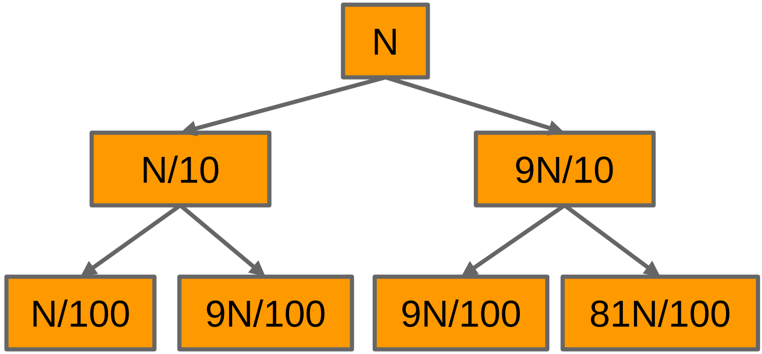
Recall that $\Theta(N \log N)$ vs. $\Theta(N^2)$ is a **really big deal**. So how can Quicksort be the fastest sort empirically? Because on average it is $\Theta(N \log N)$.

- Rigorous proof requires probability theory + calculus, but intuition + empirical analysis will hopefully convince you.

Argument #1: 10% Case

Suppose pivot always ends up at least 10% from either edge (not to scale).



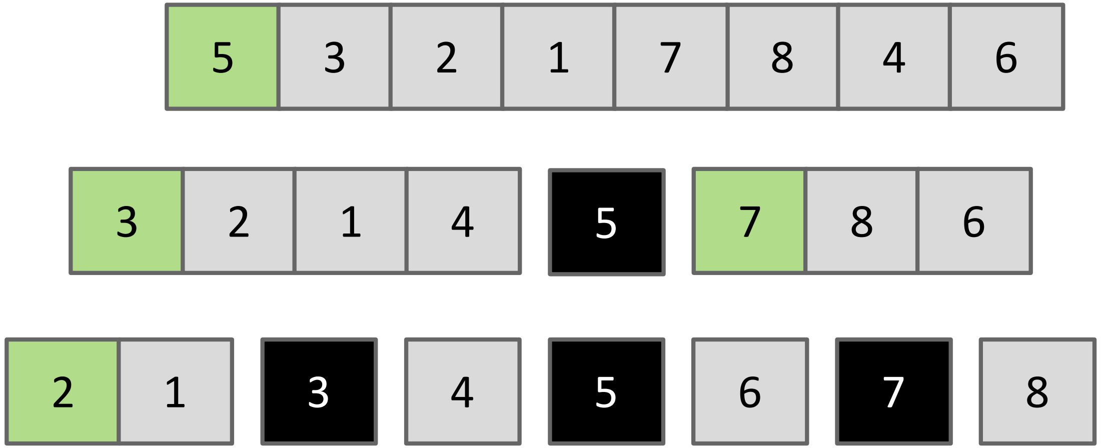


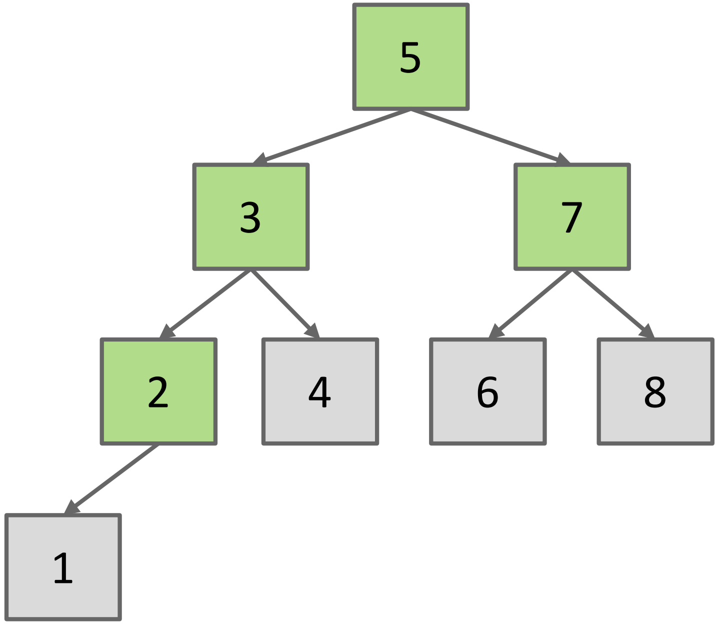
Work at each level: O(N)

- Runtime is O(NH).
    - H is approximately $\log_{10/9} N = O(\log N)$
- Overall: O(N log N).

Punchline: Even if you are unlucky enough to have a pivot that never lands anywhere near the middle, but at least always 10% from the edge, runtime is still O(N log N).

Argument #2: Quicksort is BST Sort





Key idea: compareTo calls are same for BST insert and Quicksort.

- Every number gets compared to 5 in both.
- 3 gets compared to only 1, 2, 4, and 5 in both.

Reminder: Random insertion into a BST takes O(N log N) time.

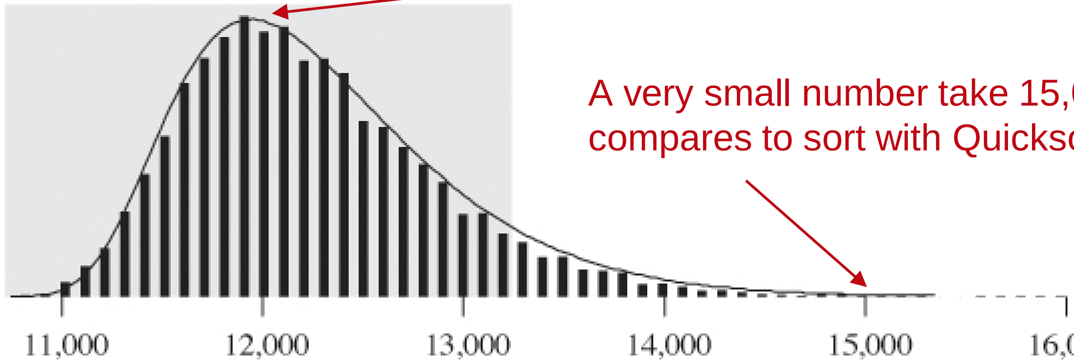
Empirical Quicksort Runtimes

For N items:

- Mean number of compares to complete Quicksort: ~2N ln N
- Standard deviation: $\sqrt{(21-2\pi^2)/3}N \approx 0.6482776N$

Lots of arrays take 12,000ish compares to sort with Quicksort.

A very small number take 15,000ish compares to sort with Quicksort.



Empirical histogram for quicksort compare counts (10,000 trials with N = 1000)

Chance of taking 1,000,000ish compares is effectively zero.

For more, see: http://www.informit.com/articles/article.aspx?p=2017754&seqNum=7

Quicksort Performance

Theoretical analysis:

For our pivot/partitioning strategies: Sorted or close to sorted.

- Best case: Θ(N log N)
- Worst case: Θ(N^2)
- **Randomly chosen array case: Θ(N log N) expected**

With extremely high probability!!

Compare this to Mergesort.

- Best case: Θ(N log N)
- Worst case: Θ(N log N)

Why is it faster than mergesort?

- Requires empirical analysis. No obvious reason why.

Sorting Summary (so far)

Listed by mechanism:

- Selection sort: Find the smallest item and put it at the front.
- Insertion sort: Figure out where to insert the current item.
- Merge sort: Merge two sorted halves into one sorted whole.
- Partition (quick) sort: Partition items around a pivot.

Listed by memory and runtime:

	Memory	Time	Notes
Heapsort	Θ(1)	Θ(N log N)	Bad caching (61C)
Insertion	Θ(1)	Θ(N^2)	Θ(N) if almost sorted
Mergesort	Θ(N)	Θ(N log N)	
Quicksort	Θ(log N) (call stack)	Θ(N log N) expected	Fastest sort

Avoiding the Quicksort Worst Case

Quicksort Performance

The performance of Quicksort (both order of growth and constant factors) depend critically on:

- How you select your pivot.
- How you partition around that pivot.
- Other optimizations you might add to speed things up.

Bad choices can be very bad indeed, resulting in $\Theta(N^2)$ runtimes.

Avoiding the Worst Case

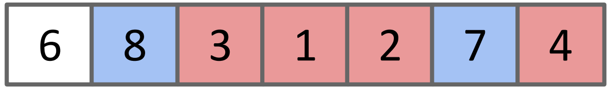
If pivot always lands somewhere "good", Quicksort is Θ(N log N). However, the very rare $\Theta(N^2)$ cases do happen in practice, e.g.

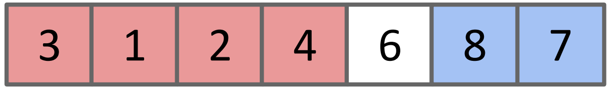
- Bad ordering: Array already in sorted order (or almost sorted order).
- Bad elements: Array with all duplicates.

What can we do to avoid worst case behavior?

Recall, our version of Quicksort has the following properties:

- Leftmost item is always chosen as the pivot.
- Our partitioning algorithm preserves the relative order of <= and >= items.





Avoiding the Worst Case: My Answers

What can we do to avoid running into the worst case for QuickSort?

Four philosophies:

1. **Randomness**: Pick a random pivot or shuffle before sorting.
2. **Smarter pivot selection**: Calculate or approximate the median.
3. **Introspection**: Switch to a safer sort if recursion goes to deep.
4. **Preprocess the array**: Could analyze array to see if Quicksort will be slow. No obvious way to do this, though (can’t just check if array is sorted, almost sorted arrays are almost slow).

Philosophy 1: Randomness (My Preferred Approach)

If pivot always lands somewhere "good", Quicksort is Θ(N log N). However, the very rare $\Theta(N^2)$ cases do happen in practice, e.g.

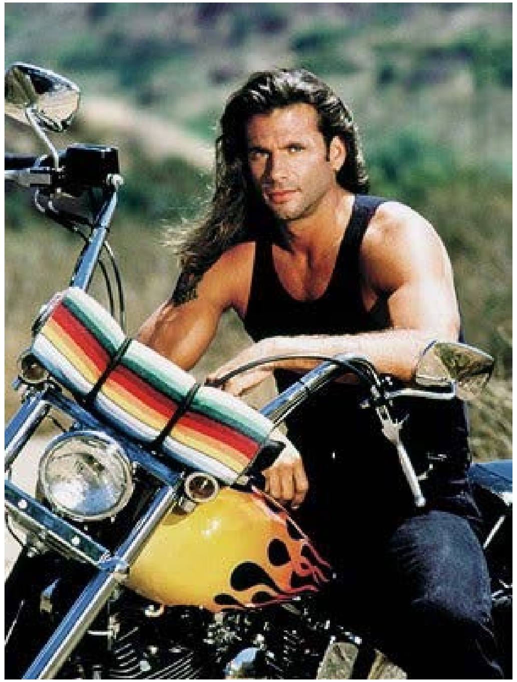
- Bad ordering: Array already in sorted order.
- Bad elements: Array with all duplicates.

Dealing with bad ordering:

- Strategy #1: Pick pivots randomly.
- Strategy #2: Shuffle before you sort.

The second strategy requires care in partitioning code to avoid $\Theta(N^2)$ behavior on arrays of duplicates.

- Common bug in textbooks! See A level problems.



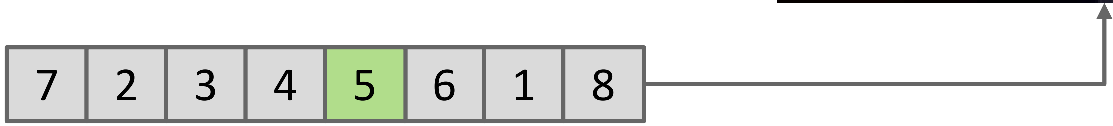
Philosophy 2a: Smarter Pivot Selection (constant time pivot pick)

Randomness is necessary for best Quicksort performance! For any pivot selection procedure that is:

- Deterministic
- Constant Time

The resulting Quicksort has a family of dangerous inputs that an adversary could easily generate.

- See McIlroy's "A Killer Adversary for Quicksort"



Dangerous input

Philosophy 2b: Smarter Pivot Selection (linear time pivot pick)

Could calculate the actual median in linear time.

- “Exact median Quicksort” is safe: Worst case Θ(N log N), but it is slower than Mergesort.

Raises interesting question though: How do you compute the median of an array? Will talk about how to do this next lecture.

Philosophy 3: Introspection

Can also simply watch your recursion depth.

- If it exceeds some critical value (say 10 ln N), switch to mergesort.

Perfectly reasonable approach, though not super common in practice.

Sorting Summary (so far)

Listed by mechanism:

- Selection sort: Find the smallest item and put it at the front.
- Insertion sort: Figure out where to insert the current item.
- Merge sort: Merge two sorted halves into one sorted whole.
- Partition (quick) sort: Partition items around a pivot.
  - Next time: Alternate strategy for partitioning, pivot identification.

Listed by memory and runtime:

	Memory	Time	Notes
Heapsort	Θ(1)	Θ(N log N)	Bad caching (61C)
Insertion	Θ(1)	Θ(N^2)	Θ(N) if almost sorted
Mergesort	Θ(N)	Θ(N log N)	
Random Quicksort	Θ(log N) (call stack)	Θ(N log N) expected	Fastest sort

Median of Medians

Median finding.

Find the median element of a set of elements: $a_1, \ldots, a_n$.

Median is value, v, where $\frac{n}{2}$ elements are less than v, if n is odd.

Versus Average?

Average household income (2004): \$70,700
Median household income (2004): \$43,200

Why so different? Bill Gates and Jeff Bezos. The 1%, perhaps.

Why use average?

Find average? Compute $\frac{\sum_i a_i}{n}$

$O(n)$ time.

Compute median? Sort to get $s_1, \ldots s_n$. Output element $s_{n/2}$.

$O(n \log n)$ time.

Better algorithm?

Solve a harder Problem: Selection.

For a set of n items S.
Select k th smallest element.

Median: select $n/2$ + 1 elt.

Example.
k = 7 for items { 11 , 48 , 5 , 21 , 2 , 15 , 17 , 19 , 15 }
Output?
(A) 19
(B) 15
(C) 21
????

Solve a harder Problem: Selection.

```
Select$(k, S)$:
    Base Case: $k = 1$ and $|S| = 1$, return elt.
    Choose rand. pivot elt $b$ from $A$.
    Form $S_L$ containing all elts $< v$
    Form $S_v$ containing all elts $= v$
    Form $S_R$ containing all elts $> v$
    If $k \leq |S_L|$, Select($k, S_L$).
    elseif $k \leq |S_L| + |S_v|$, return $v$.
    else Select($k - |S_L| - |S_v|, S_R$)
```

$k = 7$

S : 11 , 48 , 5 , 21 , 2 , 15 , 17 , 19 , 15

$v = 15$

S_L : 11 , 5 , 2

S_v : 15 , 15

S_R : 48 , 21 , 17 , 19

$7 \leq 3?$

$7 \leq 5?$

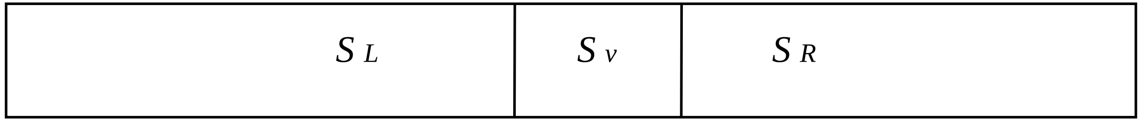
Select (2 , [48 , 21 , 17 , 19])

Will eventually return 19, which is 7th element of list.

Correctness: Induction.
Idea: Subroutine returns correct answer, and so will I !
Base case is good. Subroutine calls ..by design.

The Induction

Base Case: $k = 1$, $|S| = 1$. Trivial.



If $k \leq |S_L|$, Select(k , S_L)

k th element in first $|S_L|$ elts.

k th elt of S is k th elt of S_L

elseif $k \leq |S_L| + |S_v|$, return v ,

$k \in [\,|S_L|,\ldots, |S_L| + |S_v|]$.

k th elt of S is in S_v , all have value v

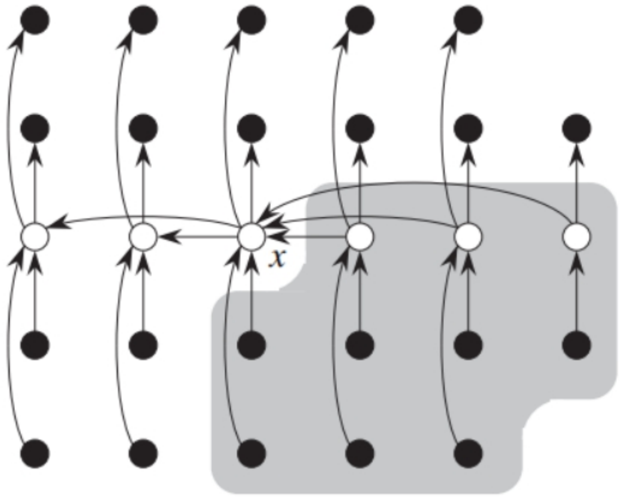
else Select ($k - |S_L| - |S_v|$, S_R)

k th element is in S_R and

k th elt of S is $k - |S_L| - |S_v|$ after elts of $S_L \cup S_v$.

Correct in all cases. $\square$

Median of Medians



- Groups of 5 elements are shown in columns.
- Medians are shown in white.
- Medians less than x are shown to the right.

Source: *Introduction to Algorithms*
Cormen, Leiserson, Rivest, Stein

- There are $n/5 * 1/2 = n/10$ medians less than x.
- For each such median, there are two more values from the group less than the median.
  - There are 3 values from each group < x.
- So there are at least $3n/10$ values < x.
- I.e. $|L| \geq 3n/10$.
- Similarly, $|R| \geq 3n/10$.

Why 5?

In general, the recursion

$$T(n) \leq T(\alpha n) + T(\beta n) + cn \ , \ T(1) = c'$$

solves to $T(n) = O(n)$ if $\alpha + \beta < 1$.

While a recursion

$$T(n) \leq T(\alpha n) + T(\beta n) + cn \ , \ T(1) = c'$$

with $\alpha + \beta \geq 1$ typically yields $T(n) = \Omega(n \log n)$.

3 does not work

Dividing the array in groups of 3 elements, we would spend $T(n/3)$ time in finding the median-of-medians.

Then, even if the size of the vector is a multiple of 3, we can only guarantee that the median-of-medians is larger than $n/3$ elements and smaller than $n/3$.

So we may recurse to a sub-array with $n - 2n/3$ elements. The recursion is

$$T(n) \leq T(n/3) + T(2n/3) + O(n)$$

No good!

Divide and Conquer

What is Divide and Conquer?

Divide a size n problem into a $(a \geq 1)$ sub-problems with size $\frac{n}{b}$ $(b > 1)$.

Template of Divide & Conquer:

Step 1: Divide the problem into sub-problems (Divide)

Step 2: Solve each sub-problem (Conquer)

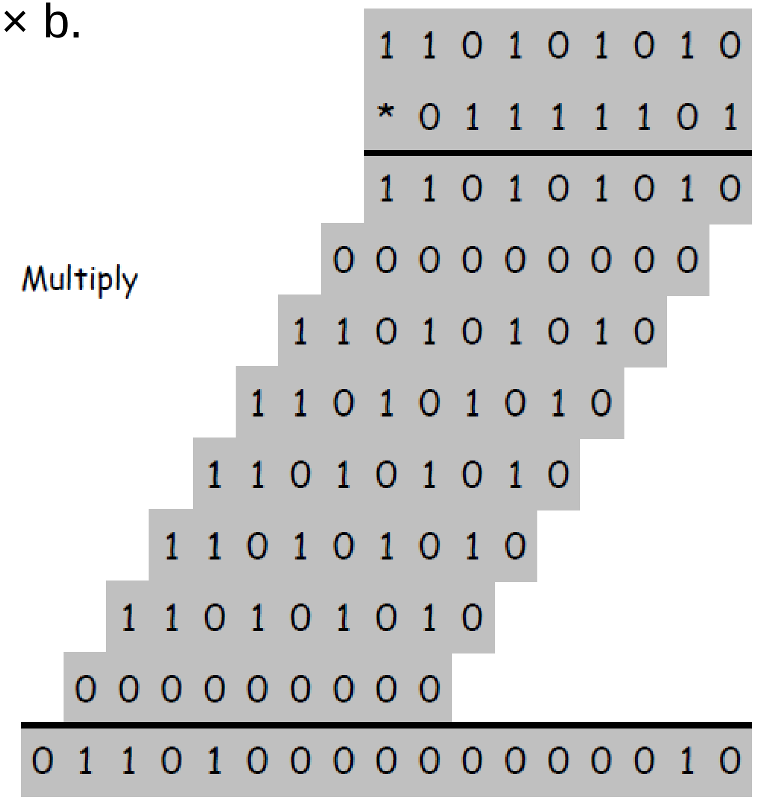
Step 3: Combine the results of sub-problems (Merge)

Example 1: Integer Multiplication

Integer Multiplication

Given two n-digit integers a and b, compute a × b.

Brute force solution: $O(n^2)$ bit operations:



```
                1 1 0 1 0 1 0 1 0
              * 0 1 1 1 1 1 0 1
                -----------------
                1 1 0 1 0 1 0 1 0
              0 0 0 0 0 0 0 0 0
            1 1 0 1 0 1 0 1 0
          1 1 0 1 0 1 0 1 0
        1 1 0 1 0 1 0 1 0
      1 1 0 1 0 1 0 1 0
    1 1 0 1 0 1 0 1 0
  0 0 0 0 0 0 0 0 0
---------------------------------
0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 1 0
```

Integer Multiplication: Method 1

Given two n-digit integers a and b, compute a × b. Using divide and conquer!

Step 1: Divide the problem into sub-problems (Divide)

$$x = \underbrace{1\,0\,0\,0}_{x_1}\ \underbrace{1\,1\,0\,1}_{x_0}$$

Step 2: Solve each sub-problem

4 sub-problems: x_0y_0, x_0y_1, x_1y_0 and x_1y_1

Step 3: Combine the result of sub-problems (Merge)

$$xy = \left(2^{n/2} \cdot x_1 + x_0\right)\left(2^{n/2} \cdot y_1 + y_0\right) = 2^n \cdot x_1y_1 + 2^{n/2}(x_1y_0 + x_0y_1) + x_0y_0$$

Compute time complexity:

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n) = O(n^2)$$

Integer Multiplication: Method 2

Given two n-digit integers a and b, compute a × b. Using divide and conquer!

Step 1: Divide the problem into sub-problems (Divide)

$$x = \underbrace{1\,0\,0\,0}_{x_1}\ \underbrace{1\,1\,0\,1}_{x_0}$$

Step 2: Solve each sub-problem

3 sub-problems: x_0y_0, x_1y_1 and $(x_0 + x_1)(y_0 + y_1)$

Step 3: Combine the result of sub-problems (Merge)

$$xy = (2^{n/2} \cdot x_1 + x_0)(2^{n/2} \cdot y_1 + y_0) = 2^n \cdot x_1y_1 + 2^{n/2}((x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0) + x_0y_0$$

Compute time complexity:

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n) = O(n^{\log_2 3}) \approx O(n^{1.59})$$

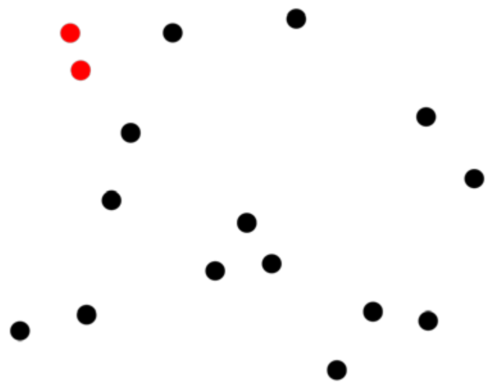
Karatsuba complexity

- $S(n) = n + (3/2)n + (3/2)2n + \cdots + (3/2)^{\lg n}n$.
- $S(n) = n\left(\frac{\left(\frac{3}{2}\right)^{1+\lg n}-1}{\frac{3}{2}-1}\right) = 2n\left(\frac{3n^{\lg 3}}{2\cdot 2^{\lg n}} - 1\right) = 3n^{\lg 3} - 2n$
  - $\lg 3 \approx 1.59$.
- So Karatsuba's method runs in $O(n^{1.59})$ instead of $O(n^2)$ time!
  - Practical for moderate n > 100.
- Useful for e.g. RSA cryptography.
- Even faster methods exist.
  - Schonhage-Strassen's Fast Fourier Transform based algorithm runs in $O(n \log n \log\log n)$ time.
  - Only practical for large n (> 10,000).

Example 2: Closest Pair of Points

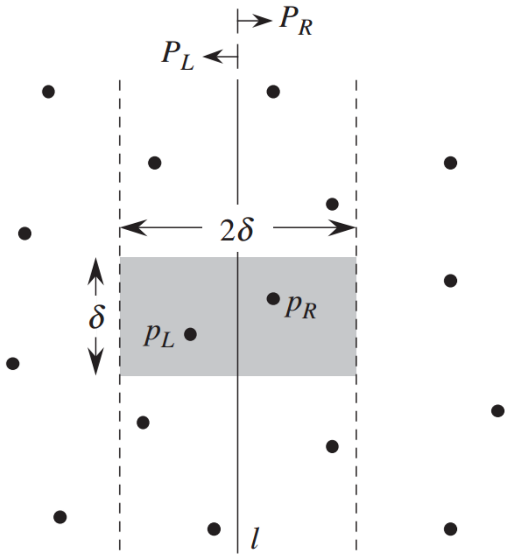
Closest point pair

- Given a set of n points in the plane, find the pair that's closest.
- Naive algorithm computes distances between all $O(n^2)$ pairs of points and chooses min.
- Use divide and conquer to improve complexity to $O(n \log n)$.



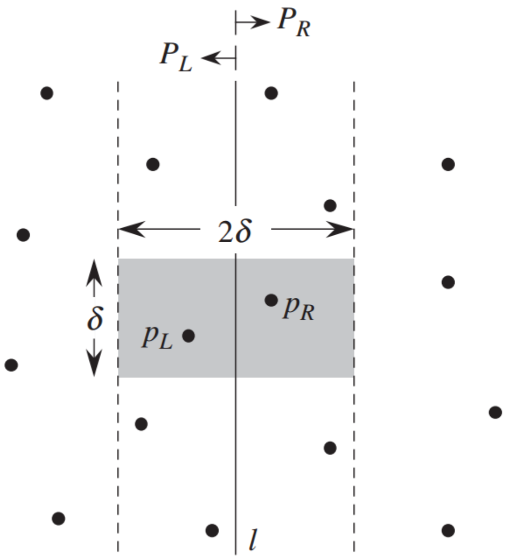
Closest point pair

- Split the points evenly using a vertical line, i.e. half the points lie on the left and half on the right.
- Observation The closest pair of points either
    - Both lie in the left half
    - Both lie in the right half, or
    - Straddles the line, i.e. one point on each side.
- This suggests the following algorithm.



Closest point pair

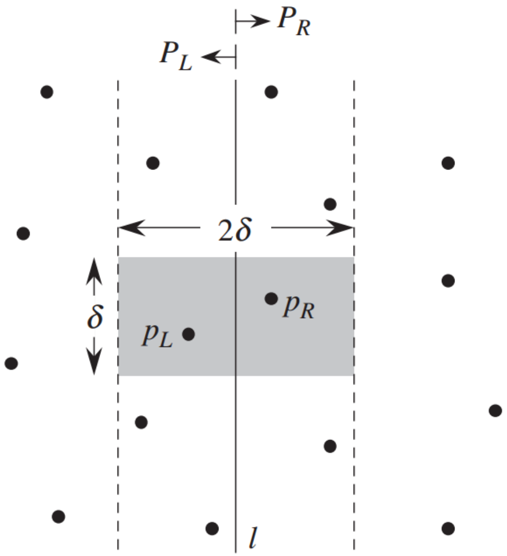
- Divide points evenly using vertical line.
- Recursively find closest point pair in left half and right half.
  - Let the min distance between any point pair in either half be δ.
- Look for closest pair of points straddling line with distance < δ.
  - Don’t need to consider straddling pairs with distance $\geq \delta$, since we already found such pairs on the left or right.
- If pair exists, return their distance.
- Else return δ.



Algorithm analysis

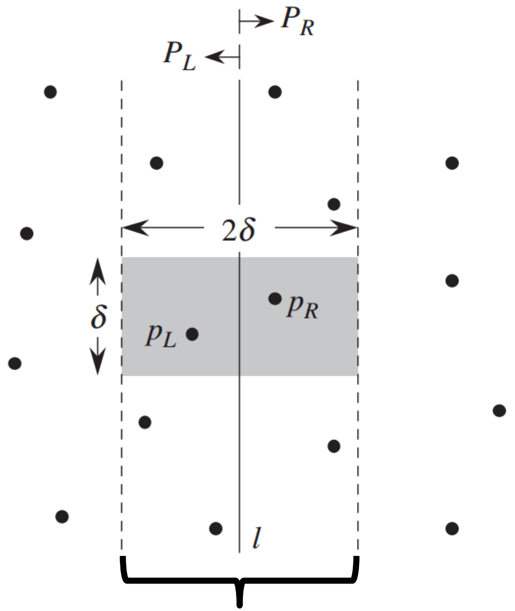
- Let $S(n)$ be time to find closest point pair of n points.
- $S(n) = 2S(n/2) + O(n)$
  - Can divide the points in $O(n)$ time.
  - $2S(n/2)$ time to recursively find closest point pair in both halfs.
  - **Can find closest straddling pair in $O(n)$ time.**
    - **Details next slide.**
- $S(2) = O(1)$.
  - If only two points, they're the closest pair.
- So $S(n) = O(n \log n)$.

- ❑ Divide the points evenly.
- ❑ Recursively find closest pair on left and right.
- ❑ Find closest straddling pair.
- ❑ Return the min of the three.



Closest straddling point pair

- Goal Find closest straddling pair, assuming their distance is $< \delta$.
- Only need to consider points within a band of width 2δ centered on dividing line.
  - Pairs outside band can't be closer than δ.
- Let B be set of points in band.
  - To form B, iterate through all points in any order, pick ones within distance δ from line.
  - Takes $O(n)$ time.
- Assume points in B sorted by y coordinate, i.e. from top to bottom.
  - By iterating in the right order when forming B, can get this property "for free", without actually sorting B.
  - Details later.
- Now, use following lemma to find closest straddling pairs.

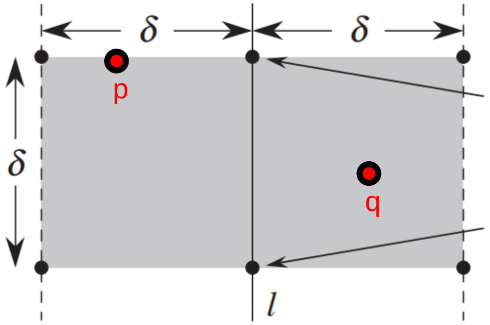


2δ band around divding line

Sparsity lemma

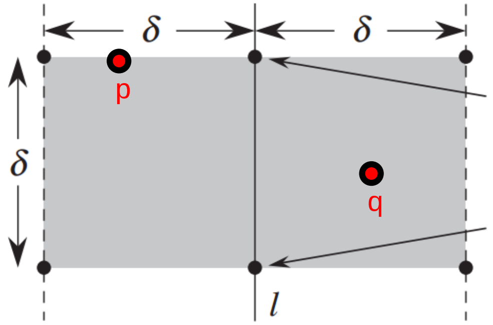
- Lemma Let $p, q \in B$. Suppose q is below p, and has distance $< \delta$ from p. Then

  1. q lies in a $\delta \times 2\delta$ rectangle centered on the dividing line, and with p on the top edge.
  2. The rectangle contains at most 6 points from B (including p and q).
  3. If we list the points in order B from top to bottom, the points in the rectangle immediately follow p in the ordering.



Sparsity lemma proof

1. Any point below the rectangle is $> \delta$ distance from p.
2. Any two points in rectangle on same side of the line are distance $\geq \delta$ apart.
   - ❖ Because δ is the min distance between any pair of points on either side.
   - ❖ So, at most 6 points in B fit in the rectangle.
   - ❖ Ex The 6 points can fit in the corners and the middle, as shown.
3. Points in the rectangle precede any points below it in y ordering.



Closest straddling point pair

- Algorithm Sweep through points in B from top to bottom.
  - For each point p, check next 5 points in R below it.
  - Let δ_p be distance to nearest one.
  - After sweeping through all points in B, return the minimum δ_p value or δ, whichever is smaller.
- Correctness By sparsity lemma, only next 5 points in B below p can be distance $< \delta$ from p.
  - Since we return the closest pair among these 5 points, we find overall closest straddling pair.
  - If no straddling pairs have distance $< \delta$, we return δ.
- Analysis Algorithm takes O(5n) time.
  - B contains O(n) points.
  - For each point in B, check its distance to 5 other points.

Dividing points evenly

- At the beginning of the algorithm, sort all points horizontally and store in an array H.
    - Takes $O(N \log N)$ time.
- Assume at some level of recursion, input array H_{sub} is sorted horizontally.
- Then points to the left / right of dividing line are points in the first / second half of array [Dividing points evenly].
    - Outputting either half takes $O(n)$ time.
- These points are sorted horizontally, for the next level of recursion.
    - So at every level of recursion, can get points in sorted order in $O(n)$ time.
- Add this $O(N \log N)$ preprocessing time to algorithm's running time.
    - Algorithm still $O(N \log N)$.

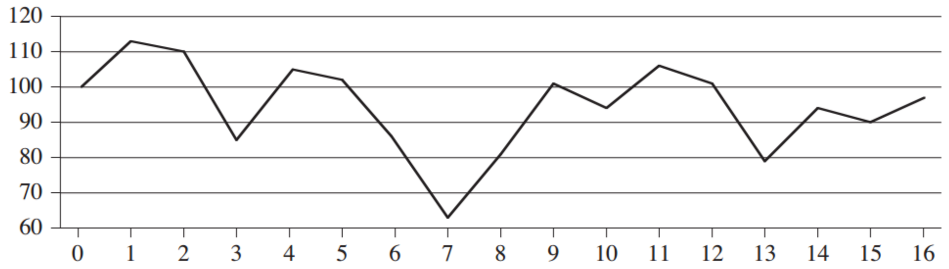
Sorting R points by y coordinate

- Assume at some level of recursion, another array V_{sub} is sorted vertically according to the input array H_{sub}.
- V_{sub} can be got by merging V_{sub}' s of two sub-problems
  - Takes $O(n)$ time.
- Points in H and V have pointers to each other.
  - i.e. given p in H, its pointer gives p's index in V. Similarly given p in V, we can get p's index in H.
- When picking out points left (or right) of dividing line using H_{sub}, mark them in V_{sub} by following the pointers.
- Next, iterate through V_{sub} (in vertical order) and pick out marked points.
  - These points are again sorted vertically.
  - Takes $O(n)$ time.
- Add this $O(N \log N)$ preprocessing time to algorithm's running time. Algorithm still $O(N \log N)$.

Example 3: Maximum Subarray

Maximum subarray

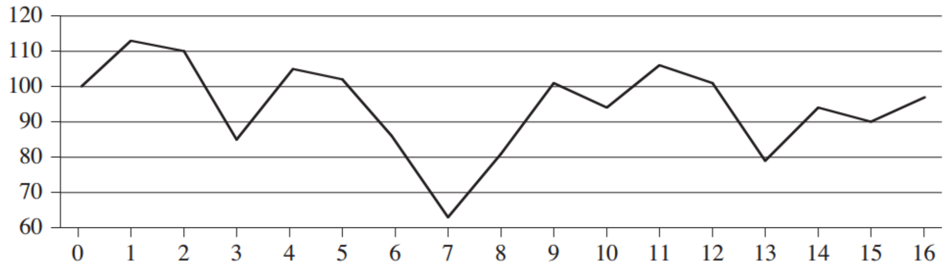
- Motivation Make money on stocks by buying and selling on days with largest price difference.



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

- Ex Buy on day 7, sell on day 11, make \$106 - \$63 = \$43.
- If there are n days, can compute price difference of all $O(n^2)$ pairs of days and take the max.
- Is there a faster way?

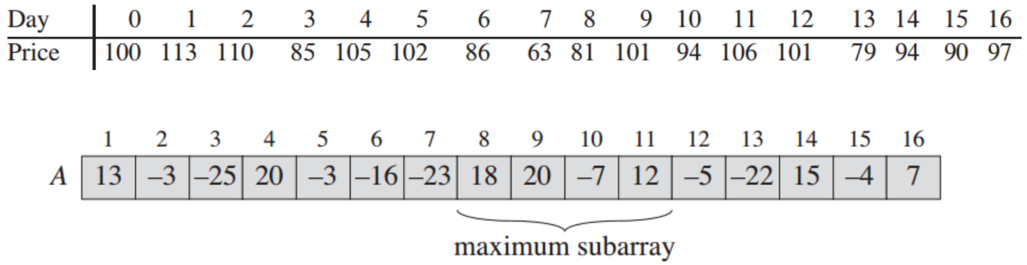
Maximum subarray



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

- Let P be the array of stock prices.
- Goal Find $i < j$ such that $P[j] - P[i]$ is maximum.
- We first compute the price change on consecutive days.
  - Ex On day 4, the price change is \$105-\$85=\$20.
  - Call the array of changes A.
  - So $A[i] = P[i] - P[i-1]$, for $i = 1, \ldots, n$.

Maximum subarray



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray

- Observation Finding $i < j$ with max $P[j] - P[i]$ is the same as finding $i < j$ with max $\sum_{k=i+1}^{j} A[k]$.
  - Ex $P[11] - P[7] = 43 = A[8] + A[9] + A[10] + A[11]$.
- Thus, we want to find a subarray of A with the maximum sum.
  - I.e. want to find a continuous set of elements of A with the largest sum.
  - Ex For A above, it's the 8th to 11th elements.

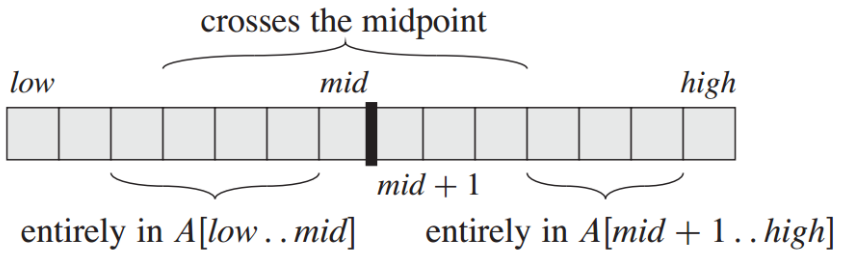
Maximum subarray

- Goal Given array A, find $i < j$ with max $\sum_{k=i+1}^{j} A[k]$.
- Seems no easier than initial problem... Still $O(n^2)$ pairs i, j to consider.
    - In fact, computing $\sum_{k=i+1}^{j} A[k]$ takes O(n) time, so finding max subarray seems to take O(n^3) time!
    - Actually, can find $\sum_{k=i+1}^{j} A[k]$ for all pairs i,j in $O(n^2)$ time. How?
- But with divide and conquer, can find max subarray in $O(n \log n)$ time.

Maximum subarray

Observation

- Divide A down the middle. Then a max subarray of A either
  - Lies entirely in the left half.
  - Lies entirely in the right half.
  - Crosses the midpoint.

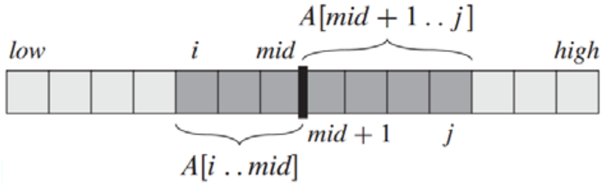


Algorithm

- Break A into left and right halves.
- Compute the max subarrays in each half.
- Compute the max subarray crossing the midpoint.
- Return max of these three subarrays.

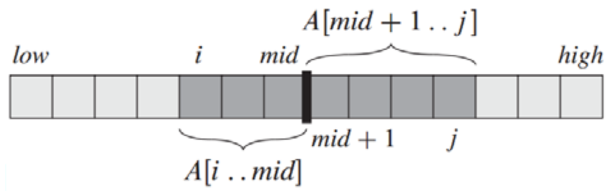
- Analysis $S(n) = 2S(n/2) + T(n) + O(1)$.
  - Finding max subarray in each half takes $S(n/2)$ time.
  - $T(n)$ = time to find max subarray crossing midpoint.
- We will show $T(n) = O(n)$.
- So $S(n) = O(n \log n)$.

Max crossing subarray



- Goal Find max subarray crossing the midpoint.
- Solution Find the max leftwards subarray from the midpoint.
  - I.e. find a subarray containing the midpoint and lying to the left, that has the max sum.
  - Also find the max rightwards subarray from the midpoint.
  - Combine them and return this.
- Ex A = [3,2,-8,1,6,7,-4,2,8,2,-4,1,-2, 3,1].
  - Max leftwards subarray from 2 is [1,6,7,-4,2].
  - Max rightwards subarray from 2 is [2,8,2].
  - Max crossing subarray is [1,6,7,-4,2,8,2].

Max crossing subarray



- Algorithm To find max leftwards subarray, sum array elements leftwards starting from midpoint.
  - Whenever sum exceeds current max, remember the index as the current max.
  - Similar for rightwards subarray.
- Analysis Scan through once to left and right. $O(n)$ time.

```
FIND-MAX-CROSSING-SUBARRAY($A$, $low$, $mid$, $high$)
1   $left\text{-}sum = -\infty$
2   $sum = 0$
3   for $i = mid$ downto $low$
4       $sum = sum + A[i]$
5       if $sum > left\text{-}sum$
6           $left\text{-}sum = sum$
7           $max\text{-}left = i$
8   $right\text{-}sum = -\infty$
9   $sum = 0$
10  for $j = mid + 1$ to $high$
11      $sum = sum + A[j]$
12      if $sum > right\text{-}sum$
13          $right\text{-}sum = sum$
14          $max\text{-}right = j$
15  return ($max\text{-}left$, $max\text{-}right$, $left\text{-}sum + right\text{-}sum$)
```

How to use divide and conquer?

How to use divide and conquer?

- The problems solved by divide-and-conquer generally have the following characteristics:
- (1) the problem can be easily solved if the scale of the problem is reduced to a certain extent
- The first feature is that most problems can be satisfied, because the computational complexity of the problem generally increases with the increase of the size of the problem.

How to use divide and conquer?

- The problems solved by divide-and-conquer generally have the following characteristics:
- (2) the problem can be decomposed into several same problems of a smaller scale, that is, the problem has the optimal substructure property.
- The second characteristic is the premise for the application of divide-and-conquer, which can be satisfied by most problems. This characteristic reflects the application of recursion.

How to use divide and conquer?

- The problems solved by divide-and-conquer generally have the following characteristics:
- (3) the solutions of the sub-problems decomposed by this problem can be merged into the solutions of this problem;
- The third feature is the key. Whether the divide-and-conquer method can be used or not completely depends on whether the problem has the third feature. If the first and second features are available, but not the third feature, greed method or dynamic programming method can be considered.

How to use divide and conquer?

- The problems solved by divide-and-conquer generally have the following characteristics:
- (4) the sub-problems decomposed by this problem are independent of each other, that is, there are no common sub-problems among the sub-problems.
- The fourth feature involves the efficiency of divide-and-conquer method. If the subproblems are not independent, divide-and-conquer method needs to do a lot of unnecessary work and repeatedly solve the common subproblems. At this time, although divide-and-conquer method can be used, dynamic programming method is generally better.