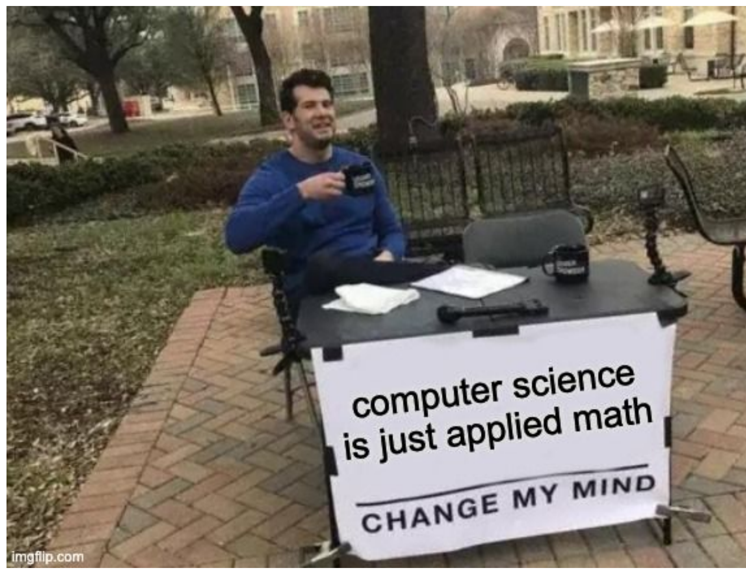


Sorting Algorithms

MSJ Computer Science Club

HAHA NO ONE'S GRADING THIS SO I DON'T HAVE TO BE "ARTISTIC" WITH DECORATED SLIDES!!!!

The proofs and all the mathematical stuff will be at the end of the lecture, so people who don’t care don’t have to listen through it

$O(n^2)$ algorithms

Selection sort: pick the smallest number and put it at the front of the array. Then the next smallest and so on. It's obvious why this works.

Bubble sort is more interesting. At each iteration, go through the array and if two adjacent numbers are out of order, swap them. Once no changes are made in an entire pass, we know it's sorted

Round 1: 3 1 4 2 -> 1 **3** 4 2 -> 1 3 **4** 2 -> 1 3 2 **4**

Round 2: 1 **3** 2 4 -> 1 2 **3** 4

Round 3: 1 2 3 4

Bubble Sort continued

We think bubble sort is irrelevant, but then this happens

USACO 2018 US OPEN CONTEST, SILVER
PROBLEM 1. OUT OF SORTS

So a couple more things about bubble sort

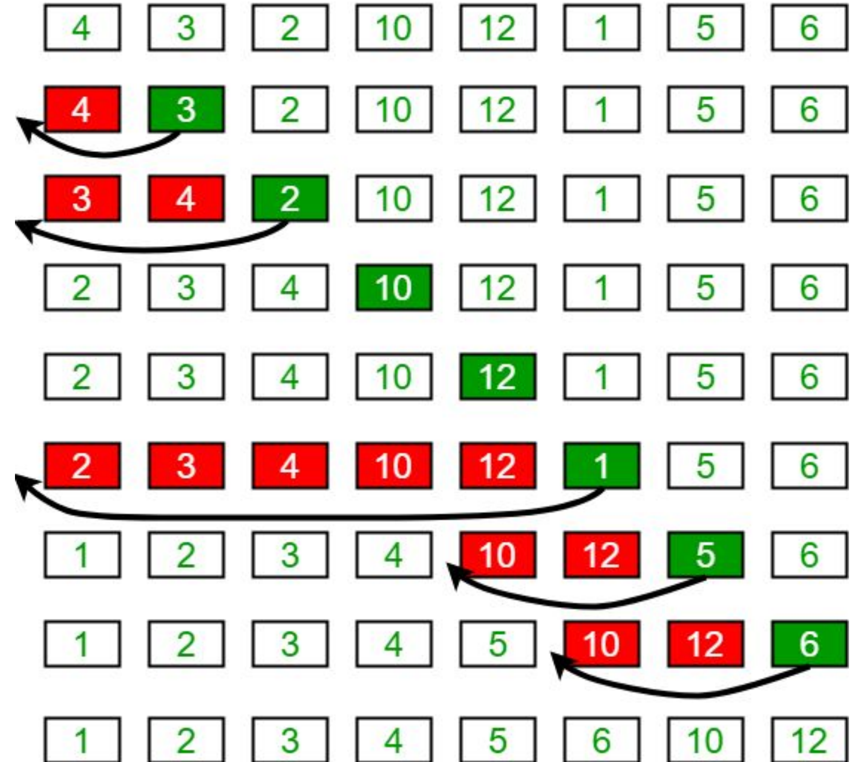
1. After the ith round, **the largest i elements are in their correct positions**: this is because at each round, the largest element moves right in every swap until it reaches its place
2. Each element can **only move left one position per swap**
3. It takes exactly **max(distance right of correct position)** rounds to sort

O(n^2) because for instance consider having 1 at the very right end and note 3

Insertion Sort

Iterate from 1 to n in the array
Compare the element to the previous element and keep going left until you get to something that’s smaller: insert it there, moving everything else up 1, so the first i elements are sorted

Binary version allows you to find the placing for it in O(log n) moves instead of O(n)



Insertion Sort Analysis

Worst case: $O(n^2)$: reverse order

Average case: $O(n^2)$: $O(n)$ moving elements each for n times

Best case: $O(n)$: already sorted

Pros

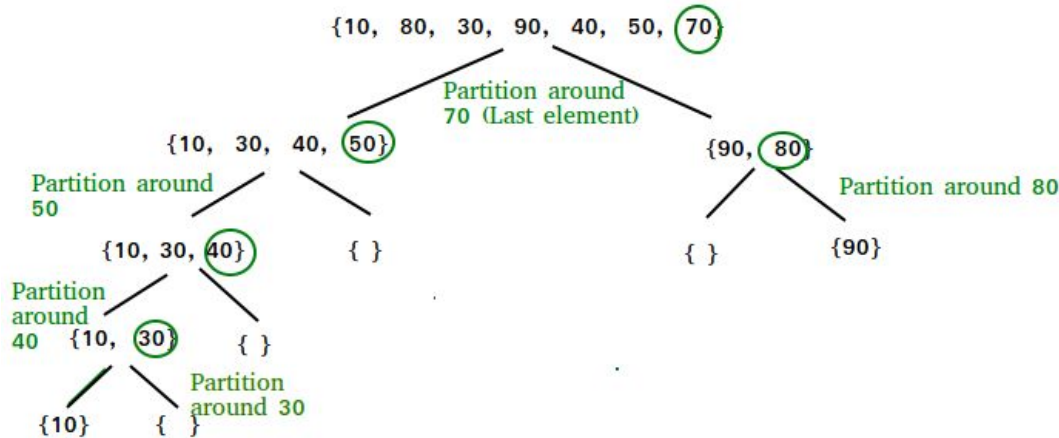
- Good when array close to sorted
- Fast for small array sizes
- $O(1)$ space

Cons

- $O(n^2)$. Nuff said

O(n log n) algorithms — quick sort

1. Choose some element as a pivot — for now, we'll go with the last element
2. Put all elements smaller than the pivot to the left of it and all elements larger than the pivot to the right of it — this is done in O(n) time and is called the **partition**
3. Call quicksort on the left side of the pivot and the right side of the pivot



Ty Geeksforgeeks for nice diagram!

Quick sort — partitioning

1. Choose the end element as the pivot, and let the part of the array that we're sorting be [low, high]
2. Go from left to right. Keep a pointer that starts at low - 1. If you see an element ≤ pivot, increment the pointer and swap the current position's element with the pointer's
3. At the end, swap the pivot element with the pointer+1 position

Current order	Compare	ptr	i
2 6 3 5 1 4	2 ≤ 4	-1	0
2 6 3 5 1 4	6 > 4	0	1
2 6 3 5 1 4	3 ≤ 4	0	2
2 3 6 5 1 4	5 > 4	1	3
2 3 6 5 1 4	1 ≤ 4	1	4
2 3 1 5 6 4	Move pivot	2+1 = 3	5
2 3 1 4 6 5			

Quick sort Analysis

Best Case: O(n log n) time

Average Case: O(n log n) time

Worst Case: $O(n^2)$ time

- This happens if the pivot is the smallest or largest element each time

Pros

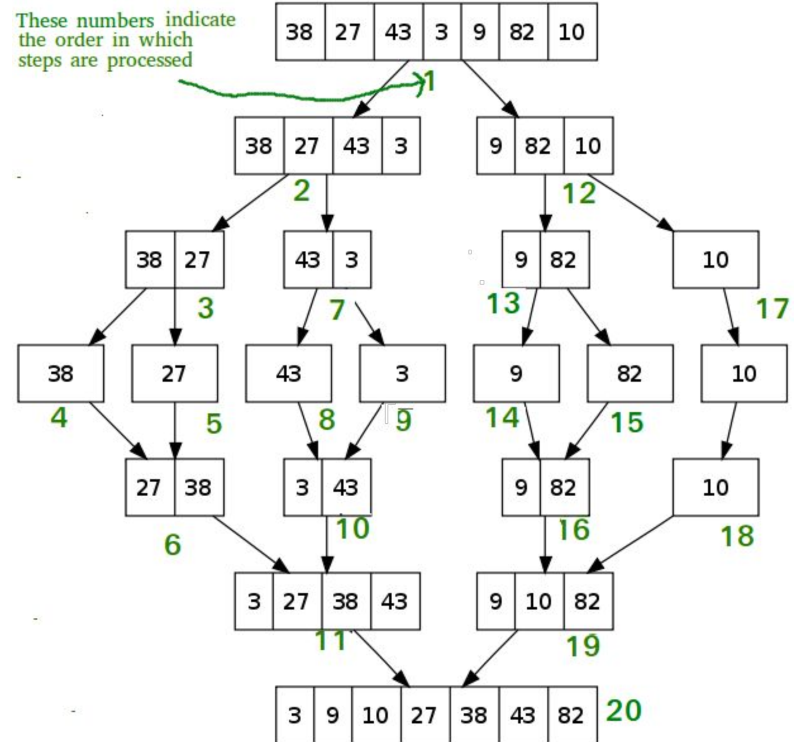
- In practice, is faster than merge sort and heap sort because “its inner loop can be efficiently implemented on most architectures”
- No extra space required
- Lots of optimizations: better pivot selection in particular https://www.techiedelight.com/boost-quicksort-performance/

Cons

- That worst case

O(n log n) algorithms — merge sort

1. Divide the array into two halves
2. Call mergesort on each half
3. **Merge the two sorted halves into one sorted part**



Merge sort — merge function

Have two pointers for the positions of each of the halves that we are on and a temporary array to store the final order

1. Compare the elements at both pointers
2. Put the smaller element into the lowest position in the temp array and increment that pointer
3. Repeat until one of the pointers reaches the end
4. Then put in all the elements of the other half until that pointer reaches the end

Temp

2 3 6 7 8 9

Left half	Right half	compare	i	j
3 7 8	2 6 9	3 > 2	0	0
3 7 8	2 6 9	3 ≤ 6	0	1
3 7 8	2 6 9	7 > 6	1	1
3 7 8	2 6 9	7 < 9	1	2
3 7 8	2 6 9	8 < 9	2	2
3 7 8	2 6 9	Add 9 to temp	3	2

Merge sort analysis

Best case: O(n log n)

Average case: O(n log n)

Worst case: O(n log n)

Pros

- Consistent
- Useful for sorting linked lists because you can insert wherever, meaning we don't need a temporary array
- inversion counting
- Used when you have large amounts of data Don't ask me why idk real life application stoof

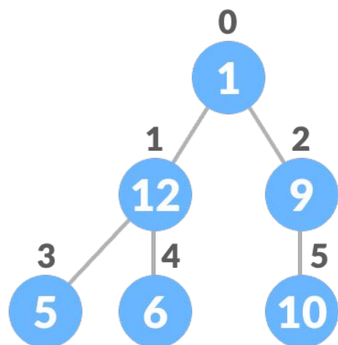
Cons

- Extra space
- Slightly slower

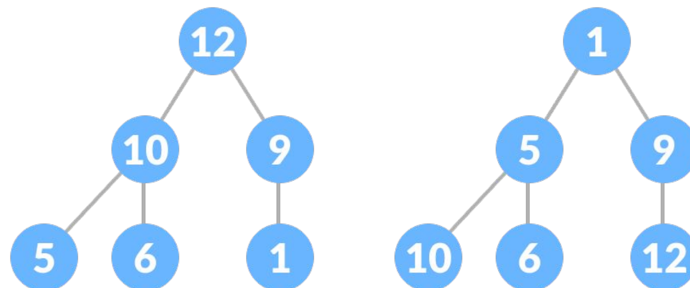
O(n log n) algorithms — Heap sort

First: a **complete binary tree** is a binary tree in which every non-leaf node has two children

A **heap** is a complete binary tree in which the parent either has a larger value than all its children or a smaller value, called max heap and min heap respectively



Complete binary tree



Max heap, min heap

Heap sort

1. Build a max heap from the given array
2. Swap the root node and the last node in the heap
3. Remove the last node / put it onto the final ordering on the left
4. **Heapify**
5. Repeat steps 2 to 4 until all the tree is just one node that we put on the final array

Heapify (a node)

Assumes that the two subtrees of the node are max heaps already

1. If the node is the largest of it and its children, don't do anything; otherwise, swap it with the largest child
2. Heapify the child's subtree

Results in a max heap: the node + its children's subtrees

Heapify (a node)

Assumes that the two subtrees of the node are max heaps already

1. If the node is the largest of it and its children, don't do anything; otherwise, swap it with the largest child
2. Heapify the child's subtree

Results in a max heap: the node + its children's subtrees

Building the max heap

```
for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);
```

<u>Heapify the non-leaf nodes going in backwards order of position</u>

This makes sure that when we heapify a node, its children and their children have already been heapified, so the subtrees are sorted

Heap sort — reminder of what the algorithm was

1. Build a max heap from the given array
2. Swap the root node and the last node in the heap
3. Remove the last node / put it onto the final ordering on the left
4. **Heapify** the root
5. Repeat steps 2 to 4 until all the tree is just one node that we put on the final array

Heap sort — a really good illustration

https://www.programiz.com/dsa/heap-sort

Heap sort analysis

Worst case: O(n log n)
Average case: O(n log n)
Best case: O(n log n)
Pros

- Consistent
- No extra space
- Easy to extract the largest or smallest few elements from the list without disturbing the remaining elements — priority queue (Fibonacci heaps though)
- Can sort nearly sorted / k-sorted arrays fast — O(k) + O((n-k)*logK)

Cons

- Worst of the 3 — bad constant factor

Introsort

used in the c++ built in sort function
Combination of quicksort, heapsort, and insertion sort

“So first it creates a partition. Three cases arises from here.

1. If the partition size is such that there is a possibility to exceed the maximum depth limit then the Introsort switches to Heapsort. We define the maximum depth limit as 2*log(N)
2. If the partition size is too small then Quicksort decays to Insertion Sort. We define this cutoff as 16 (due to research). So if the partition size is less than 16 then we will do insertion sort.
3. If the partition size is under the limit and not too small (i.e- between 16 and 2*log(N)), then it performs a simple quicksort.”

Keeps the speed of quicksort in the average time and the worst case time complexity of heap sort
Heap sort is used over merge sort because O(1) space complexity
Insertion sort is the fastest way to sort small arrays

Timsort

Used in (some of) java's and python's built in sort functions

"We divide the Array into blocks known as Run.
We sort those runs using insertion sort one by one and then merge those runs using the combine function used in merge sort.
The size of the run may vary from 32 to 64 depending upon the size of the array.
Note that the merge function performs well when size subarrays are powers of 2.
The idea is based on the fact that insertion sort performs well for small arrays."

Implementing sort in your code — c++

#include <algorithm> for the sort function

Arrays: sort(arr, arr+n); Vectors/most other containers: sort(v.begin(), v.end());

Decreasing: sort(arr, arr+n, greater<int>);

Pairs (pair<int, int>) get sorted by the first number, then the second as tiebreakers

For custom objects: make a compare function:

```
Bool comp(const thing &a, const thing &b)
{
	Return a.size < b.size
}
sort(arr, arr+n, comp);
```

Implementing sort in your code — java

Array a[]: Arrays.sort(a); Lists: Collections.sort(a);

Custom comparator: 2 methods

1. Create a Comparator class

```
Class CompInt implements Comparator {
    @override
    public int compare(Integer a, Integer b) {
            return (int)a - (int)b;
    }
}
```

2. Using a custom class, use Comparable interface

```
Class Node implements Comparable {
    int val;
    @override
    public int compareTo(Node n) {
          return this.val - n.val;
    }
}
```

Example: Arrays.sort(a, new CompInt()); //does custom sort on Integer array

Collections.sort(nodes); //where nodes is a list of Nodes uses provided comparison function

Implementing sort in your code — python

list.sort(reverse=True|False, key=myFunc)

Times to use sorting

- To traverse some values from “left” to “right” physically
    - X-coordinates
    - Time — chronological order
    - Sweepline
- To traverse some values from small to large
    - Amount of milk
    - Cost
    - Size
    - Edge length: kruskal’s
- To find the largest ____ values, which you know will give you the best output
    - Often in conjunction with greedy algorithms
- Literally when you’re doing any usaco problem

Example sorting: USACO 2018 Feb Silver Problem 1. Rest Stops

English (en)

Farmer John and his personal trainer Bessie are hiking up Mount Vancowver. For their purposes (and yours), the mountain can be represented as a long straight trail of length L meters ($1 \le L \le 10^6$). Farmer John will hike the trail at a constant travel rate of r_F seconds per meter ($1 \le r_F \le 10^6$). Since he is working on his stamina, he will not take any rest stops along the way.

Bessie, however, is allowed to take rest stops, where she might find some tasty grass. Of course, she cannot stop just anywhere! There are N rest stops along the trail ($1 \le N \le 10^5$); the i-th stop is x_i meters from the start of the trail ($0 < x_i < L$) and has a tastiness value c_i ($1 \le c_i \le 10^6$). If Bessie rests at stop i for t seconds, she receives $c_i \cdot t$ tastiness units.

When not at a rest stop, Bessie will be hiking at a fixed travel rate of r_B seconds per meter ($1 \le r_B \le 10^6$). Since Bessie is young and fit, r_B is strictly less than r_F.

Bessie would like to maximize her consumption of tasty grass. But she is worried about Farmer John; she thinks that if at any point along the hike she is behind Farmer John on the trail, he might lose all motivation to continue!

Help Bessie find the maximum total tastiness units she can obtain while making sure that Farmer John completes the hike.

INPUT FORMAT (file reststops.in):

The first line of input contains four integers: L, N, r_F, and r_B. The next N lines describe the rest stops. For each i between 1 and N, the $i+1$-st line contains two integers x_i and c_i, describing the position of the i-th rest stop and the tastiness of the grass there.

It is guaranteed that $r_F > r_B$, and $0 < x_1 < \cdots < x_N < L$. **Note that r_F and r_B are given in seconds per meter!**

OUTPUT FORMAT (file reststops.out):

A single integer: the maximum total tastiness units Bessie can obtain.

SAMPLE INPUT:

```
10 2 4 3
7 2
8 1
```

SAMPLE OUTPUT:

```
15
```

Solution

Main claim: the max occurs when Bessie stops for as long as possible at the fields with the maximum tastiness value of the fields to the right

- Basically greedy
- If she didn't stop for as long as possible on one of these fields, and instead spends more time t on a later field: we can have her spend t more time on the first field and t less time on that field and she'll still be ahead of Farmer John but she'll have gotten more tastiness

Pseudocode

Keep pairs of (position, tastiness)

Sort by position

Go from right to left and keep track of the max tastiness for that segment: if tastiness = maxtastiness, add this onto our to_visit places

Iterate through to_visit places

    Time we spend there = distance from previous place * (r_F - r_B)

    Add to the total

Print the total

Non-comparison based algorithms — Counting sort

If you have n integers in the range 1 to k

```
int frequency[k+1]  //frequency array of size k+1
For each element:
      Frequency[element] += 1

int final_array[k+1] //our sorted array, empty right now
For (i from 1 to k):
      Add on frequency[i] many i's to final_array
```

O(n+k) time, O(k) memory

1 5 3 3 7 8 2 2

Frequency:

1: 1
2: 2
3: 2
4: 0
5: 1
6: 0
7: 1
8: 1

Final_array:

1
1 2 2
1 2 2 3 3
1 2 2 3 3
1 2 2 3 3 5
1 2 2 3 3 5 7
1 2 2 3 3 5 7 8

Example: USACO 2019 US Open Contest, Gold
Problem 2. I Would Walk 500 Miles sorting the edges for Kruskal's

We have edges of length (2019201913x+2019201949y) mod 2019201997 for all 1 ≤ x, y ≤ N, N ≤ 7500

This is -(84x + 48y), which goes up to 84 * 7500 + 48 * 7500 = 990,000

We can do a counting sort: loop through the N^2 x and y values and calculate frequency, then go backwards through the frequency table in multiples of gcd(84, 48) = 12

990,000/12 = 82,500 = much smaller than the $N^2 \log N^2$ that we would normally need, so we can still use Kruskal's instead of Prim's

Topological Sort

For a DAG (directed, acyclic graph), find some ordering of the vertices such that if there is an edge from a -> b, b occurs after a in the ordering. There are multiple such orderings

Both O(V + E) time with space O(V)

Way 1: Kahn's algorithm

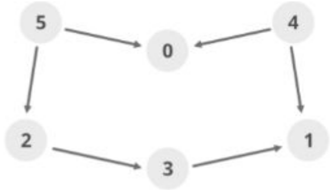
1. Find all nodes with indegree 0 and add them to a queue
2. Take a node from this queue and remove it from the graph by changing the indegrees of all its outneighbors by -1
3. Add new points with indegree 0 to the queue and repeat until empty

Way 2: dfs

Keep a visited array and a stack to store the final ordering

1. If element not visited, call dfs on it
2. In dfs
   a. Mark as visited
   b. Call dfs on non-visited neighbors
   c. Add to the stack

Topological Sort



Adja cent list (G)
0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0

	0	1	2	3	4	5
visited	false	false	false	false	false	false

Stack(empty)

Step 1: Topological Sort(0), visited[0] = true
↓
List is empty. No more recursion call.

Stack	0

Step 2: Topological Sort(1), visited[1] = true
↓
List is empty. No more recursion call.

Stack	0	1

Step 3: Topological Sort(2), visited[2] = true
↓
Topological Sort(3), visited[3] = true
↓
'1' is already visited. No more recurrsion call

Stack	0	1	3	2

Step 4: Topological Sort(4), visited[4] = true
↓
'0' , '1' are already visited. No more recurrsion call

Stack	0	1	3	2	4

Step 5: Topological Sort(5), visited[5] = true
↓
'2' , '0' are already visited. No more recurrsion call

Stack	0	1	3	2	4	5

Step 6: Print all elements of stack from top to bottom

Dfs way illustrated

The only useful algorithm — bogosort

```
while not isInOrder(deck):
    shuffle(deck)
```

An analogy for the working of the latter version is to sort a deck of cards by throwing the deck into the air, picking the cards up at random, and repeating the process until the deck is sorted.

The only useful algorithm — bogosort

Pros

- Fastest implementation
- Best case scenario is $O(n)$ time
- $O(1)$ memory
- Worst case scenario isn't $O(n^2)$
- You will never encounter the worst case scenario

Cons

- None

All good modern languages use bogosort as their main sort function and so should you

Proofs

Proof of the out of sorts problem

Helpful Claim: after each swap check in the bubble sort, the number on the right of the pair is the largest number from that point and left of the array (currently)

- the larger number of the two always goes to the right after a swap (you can formalize this with induction)

Corollary to the Helpful Claim: Any number i with a number j to the left of it and greater than it will move left in the current round

- By our helpful claim, the number swapped to the position left of i is ≥ j, so i will move left

Key claim 1: any number (call it i) with positive distance right of its correct position will move left in the current round

- If i is right of its correct position, there must be a number to the left of i that's larger, then apply the corollary

Proof of the out of sorts problem

Key claim 2: if a number (i) is in its correct position, it will never move to the right of its correct position again

- For this to happen, there would have to be an element past position i that is smaller than i, but this means there is an element before i that is greater than i (since there are i -1 positions and i-2 elements smaller than i that can be there), so apply the corollary: i will get swapped to the left instead and can't move again that round

Thus, in max(distance) turns, every number will be in a position ≤ its correct position: 1 will be in position ≤ 1 ⇒ 1 will be in position 1, 2 will be in position ≤ 2 ⇒ 2 will be in position 2, and so on.

If you want a similar problem, check out HMMT February 2018 Combinatorics Problem 7

Induction, briefly

A proof method for proving things work for all positive integers in some range

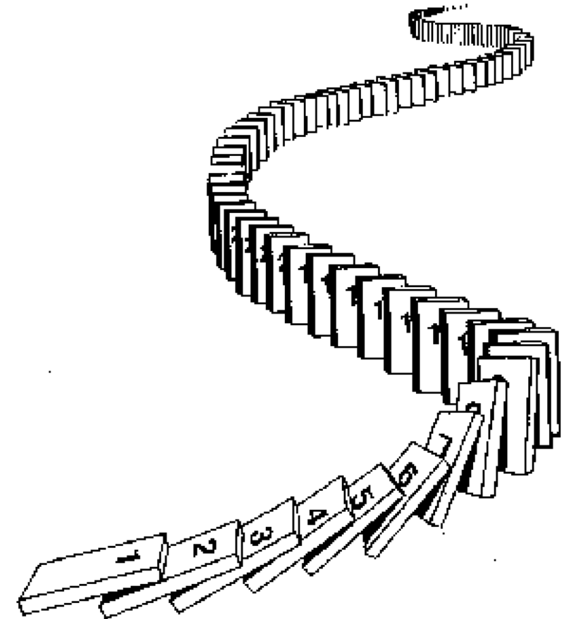
First, we have to prove that it holds true for **base case**(s), usually n = 1 (the first domino falls)

Inductive step: Then, we prove that if it's true for k = n , then it's true for k = n+1:

- Each domino knocks down the next

Strong induction is we prove that if it's true for k ≤ n, then it's true for k =n+1

- We're saying that we need all the previous dominos to have fallen down



The quintessential dominos picture

Quick sort — why it works

Partition:

- Everything ≤ the pivot is swapped to a position left of or equal to the pointer
- That means everything right of the pointer is greater than the pivot except for the pivot itself
- So we just squish the pivot between the left and right sides through that last swap

Recursive step:

- Strong induction: base case: quicksort works for $n = 1$ obviously.
- Assume it works for $k \leq n$. Then we prove it works for $n+1$
- We split into two parts and apply quicksort to each.
    - Each part gets sorted and the left part is all numbers smaller than pivot and the right part is all numbers larger than pivot: they're in sorted order too

Quick sort — time complexity

Worst case: pivot is the largest element each time, so we have O(n) partitions and each takes O(n) time for $O(n^2)$

Best case: we split in half every time (this is the most we can reduce the size by), so we have O(log n) partitions and each take O(n) time

Average case: non-rigorously, it’s just because each time we divide it’s by about 2 (and if it was like 5/4 or something change of base is a thing)

A little bit more rigorously on the next slide

Quick sort — average time complexity

Give me a sec to switch over to another window lol

https://www.overleaf.com/read/zgcjhrznrcfc

Merge sort — why it works

Merge function

The key here is that the two halves are themselves already sorted.

Claim: at each point, we take the smallest element available

- All elements left of the two pointers are already taken, since only when we take the element do we move the pointer past it
- All elements right of the two pointers are larger than at least one of the elements in the pointers (the one in its half)
- Thus the smallest remaining element is the smaller of the two at the pointers

Proving that the overall algorithm works is similar: we use induction and since mergesort works on $k = n/2$, the two halves are sorted and the merge results in a sorted final array

Heap sort — why it works

- Heapifying V resulting in a max heap
  - Induction on the size of the tree: base case is just 1 node, obvious
  - Inductive step: the max thing applies to V and its two children once we do the swap, and by the inductive hypothesis the subtree of the child we swap with will be made into a max heap
- Building the max heap
  - When you heapify V, its children have already been heapified, thus the subtrees are max heaps as required
- The algorithm itself
  - The root is the largest element in a max heap (can prove by induction — larger than children who are the largest in their subtrees)
  - After we swap the last element in and get rid of it, the two subtrees are still max heaps
  - Heapifying the root thus makes the whole thing a max heap again

Heap sort — Time complexity

Heapifying a node takes O(log n) time in the worst case because we may need to do one swap per level of the binary tree

When building, we heapify for n/2 nodes, so n/2 * log n = O(n log n)

For the sorting part, we heapify n times, for n * log n = O(n log n) time

Since these two parts are done in sequence, overall is O(n log n)

(you can probably see now why the constant factor is poor)

Kahn’s algorithm — why it works

A DAG G has at least one vertex with indegree 0 and one vertex with outdegree 0.

- Let the longest path in G go from u to v. We know this exists because there are no cycles. If u does not have indegree 0, then we could’ve enlarged the path, since the neighbor couldn’t be anything already on the path or we would have a cycle. If v does not have outdegree 0, we can similarly enlarge the path

After we remove a node, the graph will remain a DAG — you can’t create cycles by removing edges

Thus at each point in the algorithm we will be able to find an appropriate node and remove it

We get the a -> b means a goes before b condition from the fact that if something has indegree 0, it means all of its in-neighbors have already been added

Dfs for topo sort — why it works

When we do dfs for an element V, all its neighbors

1. Have already been visited and so are before it in the stack
2. Get dfs called for them. Before this dfs call returns to V, this neighbor gets added to the stack

Thus by the time we add V to the stack, its neighbors are already all in it

Proof that comparison algorithms cannot be better than O(n log n) time

http://www.bowdoin.edu/~ltoma/teaching/cs231/fall04/Lectures/sortLB.pdf

Thanks for listening!

Now I’m tired and hungry