Scientific Computing

Feb 12, 2025

Announcements

- → HW 2 due Mon, Feb. 17
- → Office hours this Friday are rescheduled to 2pm - 3pm.
- → Next Monday, no in-person lecture and no office hours.

Today

- → Search Spaces and Brute Force
- → Divide and Conquer

Office Hours:

Mon + Fri

9:30am - 10:30am

Cudahy 307

GPU Problem from HW 2:

What really is each configuration you're checking made up of? You have 60 transaction slots, and you need to assign a person to each of them. How many possibilities?

Slot 1: n people → n possibilities

Slot 2: n-1 people → n-1 possibilities

Slot 3: n-2 people → n-2 poss.

⋮

Slot 60: n-59 people → n-59 poss.

(order matters!)

All tuples of size 60 with no repeats

solution = $(P_1, P_2, P_3, \ldots, P_{60})$

Search space: all ordered lists of 60 people

Size: $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-59) = n^{60} + ?n^{59} + ?n^{58} + \ldots$
$= O(n^{60})$.

Good news: Polynomial, not Exponential!

Bad news: Yikes...

NFL Schedules: search space for 1 week = all ways of putting 32 teams in pairs. For 17 weeks = all ways of picking 17 one-week schedules

$\approx 6.5 \times 10^{294}$

(ignoring bye weeks)

# of atoms in the universe $\approx 10^{80}$

Summary of Brute Force

Pros – very easy to code
fewer bugs
guaranteed optimal
find all solutions *
* good to test other methods against

Cons – SLOW, can usually only do small cases
~> weighted interval / knapsack (2^n)
n up to 20 – 30 in a few minutes
~> pairs of points $O(n^2)$
n ≈ 100,000 in a minute (not bad!)

Comparing with Greedy Algos

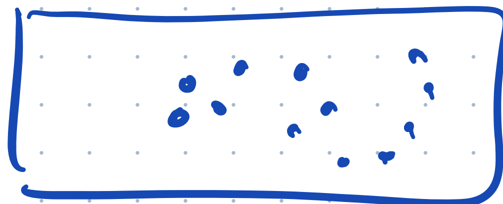
Greedy	Brute Force
Considers only 1 solutions	Considers every element
Fast	Slow
Not guaranteed optimal	Guaranteed Optimal

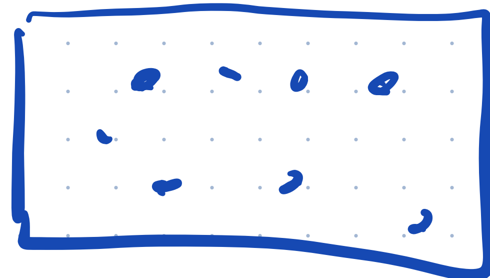
So, how can we find optimal solutions?

(1) Don't even bother — greedy algos

(2) Wander around the search space randomly, keeping track of the best you've seen.
(random search)

(3) Wander around the search space cleverly, keeping track of the best you've seen.
(metaheuristics)





So, how can we find optimal solutions?
or optimal-ish

(4) Check everything in the search space one-by-one (brute force)

(5) Check or otherwise rule out everything in the search space (divide-and-conquer, backtracking, branch-and-bound).

(6) Do some clever computations that allow you to score big chunks of the search space all at once (dynamic programming).

So, how can we find optimal solutions?

(4) Check everything in the search space one-by-one (brute force)

(5) Check <u>or otherwise rule out</u> everything in the search space (divide-and-conquer, backtracking, branch-and-bound).

(6) Do some clever computations that allow you to score big chunks of the search space all at once (dynamic programming).

Topic 6 - Divide and Conquer

"Divide and Conquer" is an algorithmic paradigm that is roughly

1) Split the input in half
2) Solve the problem on each half separately (recursively)
3) Combine your two answers into one big answer.

Classic Example: Sorting a list (easy)

* You can phrase this as a constraint satisfaction problem.
* Input: n numbers
* Search space: All orderings of n things. These are called permutations, and the # of them is
  $$n(n-1)(n-2)(n-3)\cdots 3\cdot 2\cdot 1 = n!$$
* Goal: Find the rearrangement that puts things in the right order.

* Obvious optimal algorithm: (greedy-ish)
    - Pick the smallest thing, put it first
    - Pick the next smallest thing, put it second, etc.

"insertion sort"

n items

How many steps does this take?
- Finding the kth smallest thing takes n steps (have to search the whole list)

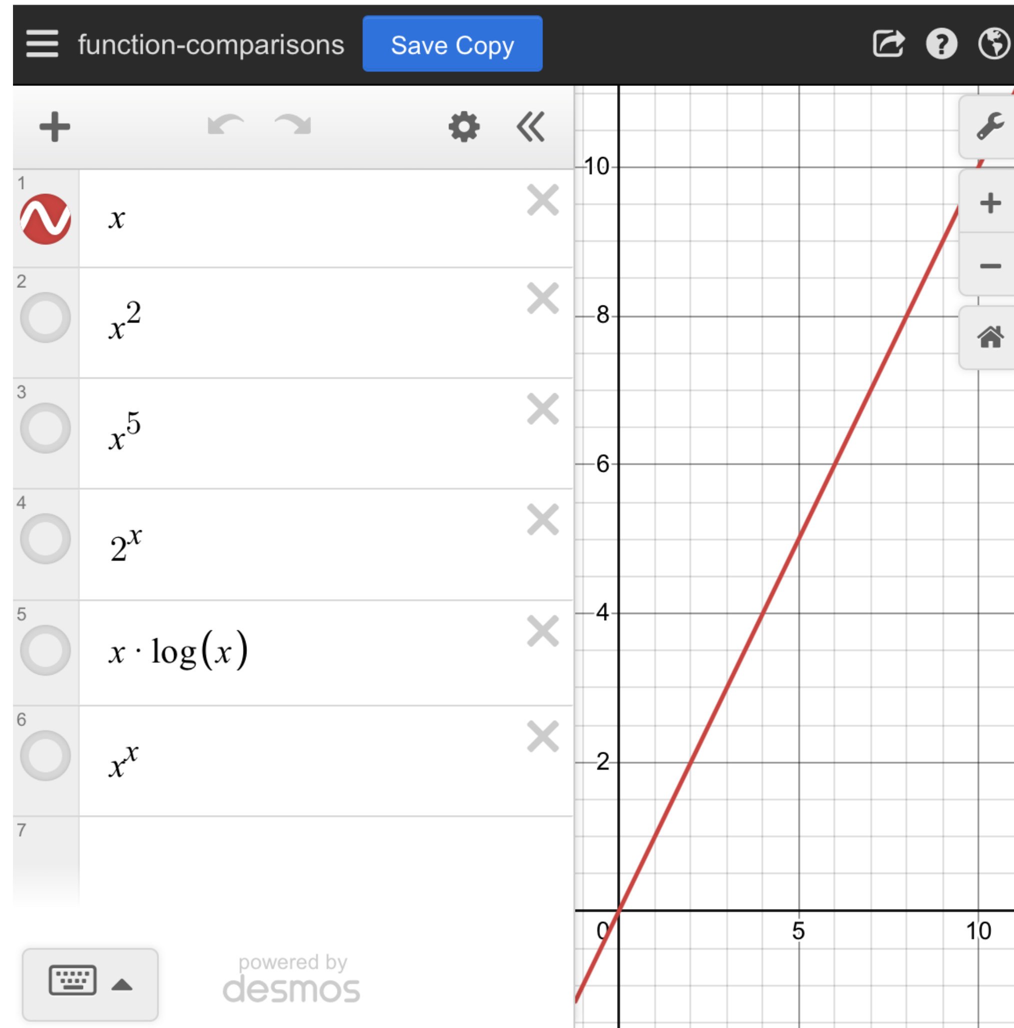
$n+(n-1)+(n-2)+\dots+1$

$\frac{n(n+1)}{2}$

- We have to do this n times.

Thus, $O(n^2)$. Fine for a few thousand things, but not more.

* Divide-and-conquer can do it in $O(n \log(n))$.



1) Split your input elements in half (or close enough)

2) Sort each half (recursively, by dividing and conquering)

3) Combine the two sorted halves into one big sorted list

Ex: Input: sort(3 19 -7 2 1 6 0 -10)

sort(3 19 -7 2)

sort(1 6 0 -10)

sort(3 19) sort(-7 2) sort(1 6) sort(0 -10)

= = = =

3 19 | -7 2 | 1 6 | -10 0

-7 2 3 19

-10 0 1 6

-10 -7 0 1 2 3 6 19

Pseudocode

```
function merge_sort(Q):        Q = list of #s
  if |Q| = 1:
      return Q
  L = left half of Q    } divide input into
  R = right half of Q   }   two parts, left & right
  L = merge_sort(L)
  R = merge_sort(R)
```

$|S|$ = # of things in S

len(S)

```
new_list = []
while |L|+|R| ≥ 0:
   take L[0] or R[0], whichever is smaller,
      remove it, and add to new_list
return new_list
```

What's the runtime? Harder, because it's recursive. What we <u>can</u> do is find a <u>recurrence</u> for the runtime.

Suppose the runtime is $T(n)$ when the input has size n.

$n/2$ is the size of the left half

Steps:

Apply to left half $T(n/2)$

Apply to right half $T(n/2)$

Merge

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

Recurrence: $T(n) = 2T\left(\frac{n}{2}\right) + n$

There is a theorem called The Master Theorem that tells you how to convert a recurrence into a formula.

$T(n) = 2 \cdot T(\frac{n}{2}) + n$

In this case, it tells us: merge_sort

$\Rightarrow$ $T(n) = O(n \log(n))$.

$\rightsquigarrow$ Jupyter Notebook Sorting demo