Complexity Analysis – Time vs. Space

Ziyi Cai

2022/10/31

Searching in Unordered Lists

- Given an unordered list $\mathbf{L}$ of n elements and a search key k.
- We seek to identify one element in $\mathbf{L}$ which has key value k, if any exists.
- *For ease of discussion, we will assume that the key values for the elements in $\mathbf{L}$ are unique.*

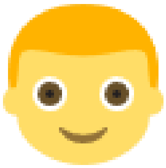
- A simple brute-force search will suffice.
- In any cases, at most n comparisons are needed.
- In worst cases, n comparisons($<$,$>$,$=$) are needed.

- Can we do better than this?
- **Unfortunately, the answer is NO!**

```
bool flag = false;
for (int i = 0;i < n;++ i)
  if (L[i] == K) {
    flag = true;
    printf("Found!\n");
  }
if (!flag)
  printf("Not found!\n");
```

Lower Bound on Searching in Unordered Lists

Claim.
The lower bound for the problem of searching in an unordered list is n comparisons.

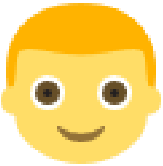


- Proof by contradiction.
- Assume an algorithm A exists that requires only $n-1$ (or less) comparisons of k with elements of $\mathbf{L}$, A must have avoided comparing k with $\mathbf{L}[i]$ for some value i.
- We can feed the algorithm an input with k in position i.
- Then the result of A is incorrect!

- Wait a minute, something is wrong here!
- Any given algorithm need not necessarily consistently skip any given position i in its $n-1$ searches.
- It is not even necessary that all algorithms search the same $n-1$ positions first each time through the list!

Lower Bound on Searching in Unordered Lists

Claim.
The lower bound for the problem of searching in an unordered list is n comparisons.



- Hmmm… ok, let me fix this.
- On any given run of the algorithm, *some* element position (call it position i) gets skipped.
- It is possible that k is in position i at that time, and will not be found.
- I am still a bit confused.
- Why should we always compare elements of $\mathbf{L}$ against k?
- An algorithm might make useful progress by comparing elements of $\mathbf{L}$ against each other!

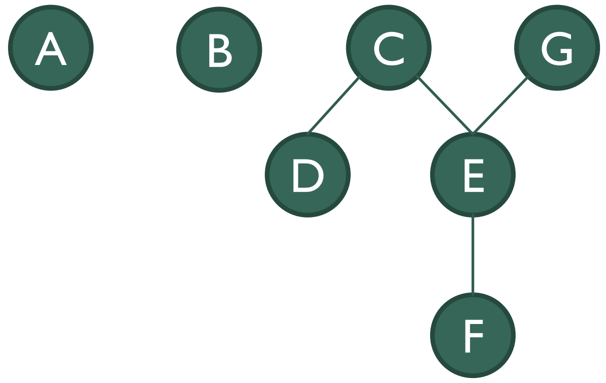
- Great! This proof seems quite convincing.
- Such comparisons won't actually lead to a faster algorithm, but … how do we know for sure?

Order Theory

- *"Definition". A **total order** defines relationships within a collection of objects such that for every pair of objects, one is greater than the other.*
  - The letters of the alphabet ordered by the standard dictionary order, e.g., $A < B < C$, etc., is a strict total order.
  - The set of real numbers ordered by the usual "less than or equal to"($\leq$) or "greater than or equal to"($\geq$) relations is totally ordered.
- *"Definition". A **partially ordered set** or **poset** is a set on which only a partial order is defined.*
  - The set of natural numbers equipped with the relation of divisibility is a partial order.

Lower Bound on Searching in Unordered Lists

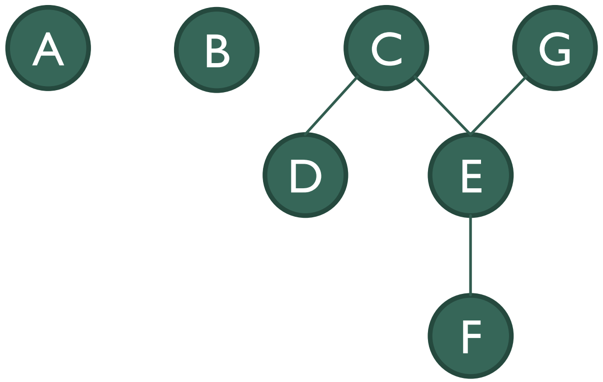
- For our purpose here, the partial order is the state of our current knowledge about the objects.
- We can represent this knowledge by drawing directed acyclic graphs showing the known relationships.



Lower Bound on Searching in Unordered Lists

Theorem.
The lower bound for the problem of searching in an unordered list is n comparisons.

- comparison between elements in $\mathbf{L}$
  - **at best** combine two of the partial orders together
  - after m comparison of this type, **at least** $n - m$ posets remain
- comparison between k and an element in $\mathbf{L}$
  - each poset requires **at least** one comparison of this type to make sure that k is not somewhere in it
- Thus, any algorithm must make **at least** $n - m + m = n$ comparisons in the worst case.



Lower Bound on Searching in Ordered Lists*

- A binary search will suffice.
- In worst cases, $O(\log n)$ comparisons are needed.

```c
int l = 0, r = n - 1;
for (int mid;l <= r;) {
  mid = l + r >> 1;
  if (L[mid] < k)
    l = mid + 1;
  else
    r = mid - 1;
}
if (L[l] == k) printf("Found!\n");
else printf("Not found!\n");
```

- Can we do better than this?
- The answer is NO!
- An argument using decision tree can show that any algorithm on an ordered list requires at least $\Omega(\log n)$ comparisons in the worst case.

- For the stated reason, binary search is *the optimal algorithm* on searching in ordered lists.

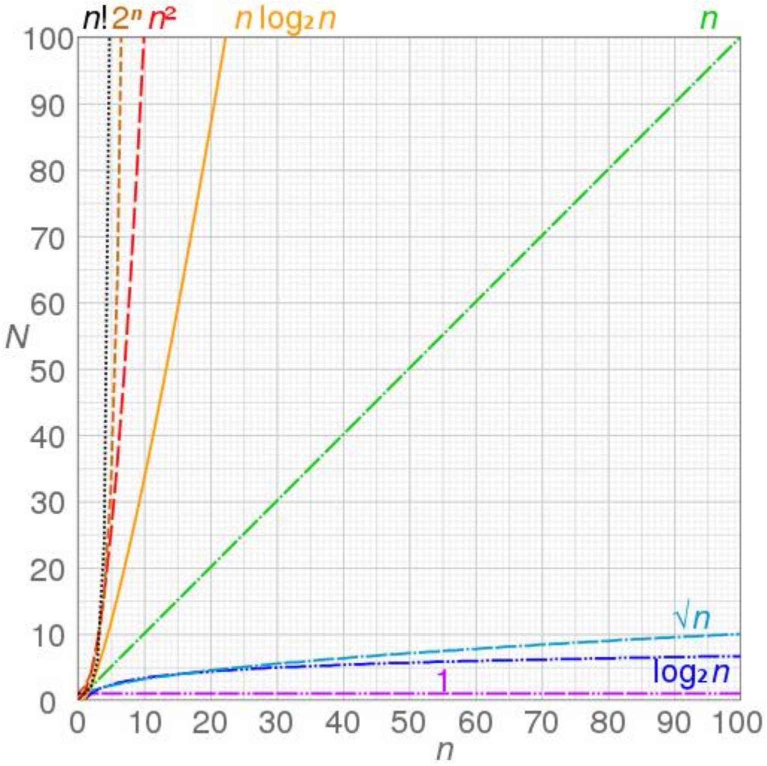
“Algorithm for Designing Algorithms”

- What does the lower bound of a problem tell us?
- What does the upper bound of a problem mean?

- Putting together all that we know so far about algorithms, we can constantly improve our algorithm until we are satisfied or exhausted.

Introduce to Complexity Analysis

- Do constant factors matter?
  - Asymptotic notation
    - $O(f(n)), o(f(n)), \Omega(f(n)), \Theta(f(n))$
- Which scenario should we focus on?
  - the best cases
  - the average cases
  - the worst cases



Back to Turing Machine

- How to measure the resource used by a Turing machine?
- time
  - the steps taken by the Turing machine
- space
  - the number of locations ever visited on the word tapes*

Measure of Time and Space

- The class $\mathbf{TIME}\big(T(n)\big)$ or $\mathbf{DTIME}\big(T(n)\big)$
  - A language L is in $\mathbf{TIME}\big(T(n)\big)$ iff there exists a TM $\mathbb{M}$ that runs in $cT(n)$ time and decides L.
- The class $\mathbf{SPACE}\big(S(n)\big)$ or $\mathbf{DSPACE}\big(S(n)\big)$
  - A language L is in $\mathbf{SPACE}\big(S(n)\big)$ iff there exists a TM $\mathbb{M}$ that runs in $cS(n)$ space and decides L.

Universal Turing Machine, Revisited

Theorem. (Hennie and Stearns, 1966)
There is a universal TM $\mathbb{U}$ that $\mathbb{U}(x, \alpha)$ halts in $cT(|x|) \log T(|x|)$ steps if $\mathbb{M}_\alpha(x)$ halts in $T(|x|)$ steps, where c is a polynomial of α.

Theorem.
There is a universal TM $\mathbb{U}$ that operates without space overhead for input TM with space complexity greater than $\log n$.

Does computational models matters?

Cobham-Edmonds Thesis.
Every "reasonable"(physically realizable) model of computation can be simulated by a Turing machine with only a polynomial slowdown.

- Possible counterexamples?
  - randomized computation
  - parallel computation
  - quantum computation

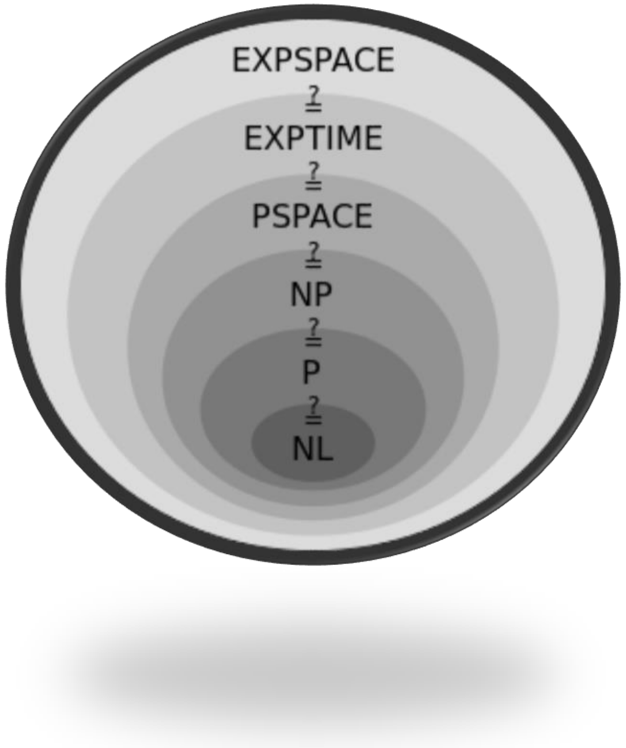
Gödel's Lost Letter(1988)

If there really were a machine with $\varphi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem.

-Kurt Gödel, 1956

Time and Space Resources

- Time complexity classes
  - $\mathbf{P} \coloneqq \bigcup_{i=1}^{\infty} \mathbf{TIME}(n^i)$
  - $\mathbf{EXP} \coloneqq \bigcup_{i=1}^{\infty} \mathbf{TIME}\left(2^{n^i}\right)$
- Space complexity classes
  - $\mathbf{L} \coloneqq \mathbf{SPACE}(\log n)$
  - $\mathbf{PSPACE} \coloneqq \bigcup_{i=1}^{\infty} \mathbf{SPACE}(n^i)$
  - $\mathbf{EXPSPACE} \coloneqq \bigcup_{i=0}^{\infty} \mathbf{SPACE}\left(2^{n^i}\right)$
- $\mathbf{L} \subseteq \mathbf{P} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}$



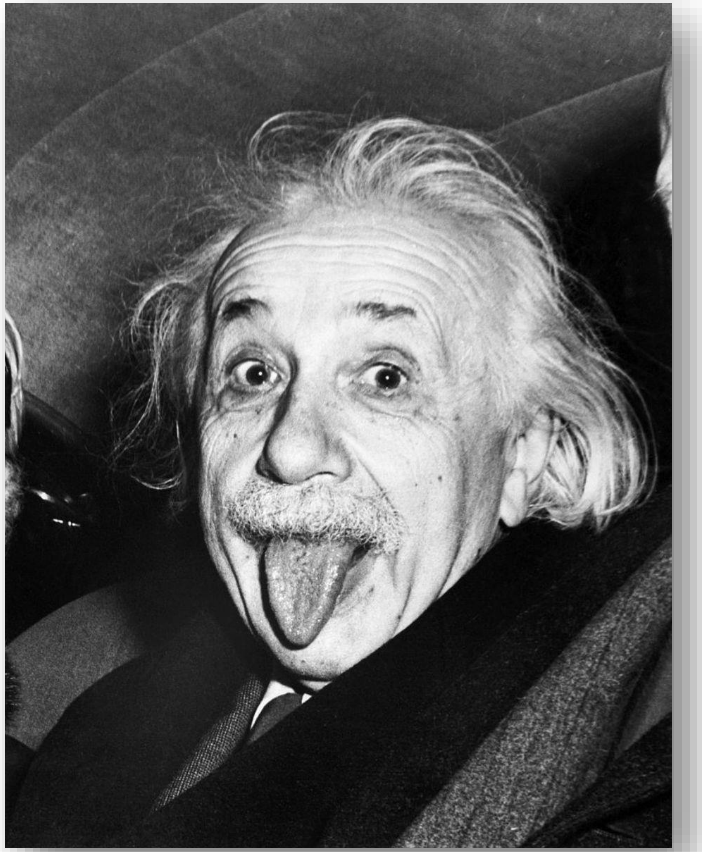
Time-space Tradeoff in Practice

- compressed vs. uncompressed data
  - compressed: easy to store, takes extra time to decompress
  - uncompressed: easy to process, takes extra memory to store
- brute-force search vs. lookup table
- cache
- meet-in-the-middle attack

A soft question: is time and space interchangeable?

Hopcroft-Paul-Valiant Theorem. (Hopcroft, Paul and Valiant, 1975)
For all space constructible $S(n)$, $\mathbf{TIME}\big(S(n)\big) \subseteq \mathbf{SPACE}(S(n)/\log S(n))$.

- Problems remain open:
  - $\mathbf{L} \stackrel{?}{=} \mathbf{P}$
  - $\mathbf{PSPACE} \stackrel{?}{=} \mathbf{EXP}$
- Do we have a theory about time vs. space like Physics?



THANKS FOR LISTENING.