We cut the algo into 2: f o g , such as f gives the min info(min number of bits) to g

then C(g)=O(1) because it’s just a mapping, and Cworst(f) >= #bits, #bits = lg(min number of brute force cases)

* That’s independent of what data structure is used

Sorting: Mapping of g() for n=3

|  |  |  |  |
| --- | --- | --- | --- |
| **a<b** | **b<c** | **a<c** | **ret** |
| 1 | 1 | 1 | abc |
| 1 | 0 | 1 | acb |
| 0 | 1 | 1 | bac |
| 1 | 0 | 0 | cab |
| 0 | 1 | 0 | bca |
| 0 | 0 | 0 | cba |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | 1<2 |  |  |  |  |  |  |
|  |  | 2<3 |  |  |  |  |  |  |  | 2<3 |  |  |
| cba |  |  |  | 1<3 |  |  |  | 1<3 |  |  |  | abc |
|  |  |  | bca |  | bac |  | cab |  | acb |  |  |  |

Can you generalize this?

Sorting has n! different results, binary decision tree has to have n! leafs. So the height of tree is lg(n!)

Another way: What can be the min output of sort function? Sort takes n! different inputs (lg(n!) bits) and min size of output is lg(n!)

For a search algorithm it is normal that the algo has to check each case of the input. So the time complexity is at least input size (log of number of cases). Once input size is known (i.e. nlg(n)) that gives the structure of the search algo. In case it’s the min input size, you have the best time complexity.

Time complexity depends primarily on input size.

While sorting n boxes I have n! different situations. I give to g the situation number (which occupies lg(n!) bits). G has a mapping of situation number ⬄ arrange operations. G’s output (arrange operations) occupies lg(n!) bits.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | #cases | Theo lower bound | Implemented lower bound | Test complex |
| Sorting n boxes | n! | Cworst(sort) >= lg(n!)  lim_(y->1^-) lim_(x->(+∞)^-) (log(2, x!))/(x log(2, x)) = 1 | nlg(n) | n-1 |
| Finding max of n boxes | n | Cworst(merge) >= lg(n) | n | n-1 |
| Merging 2 sorted lists of n and m boxes | combin(n,m) | Cworst(merge) >= lg(combin(n,m)) | n+m-1 | n+m-1 |
| Merging 2 sorted lists of n boxes | combin(2n,2)  f(1)=2, f(2)=6, f(3)=20 | lg(combin(2n,n))  lim_(y->1^-) lim_(x->(+∞)^-) (log(2, binomial(2 x, x)))/(2 x - 1) = 1 | 2n-1 | 2n-1 |
| Inserting a box to a sorted list of n | n+1 | Cworst(insert)>=lg(n+1) | n | n-1 |
| TSP | (n-1)!/2, no! this #iterations to brute force search  ((n-1)n/2)! |  | n^2 . 2^n |  |
| Find if a sorted list contains a duplicate | n-1 |  | n-1 |  |
| BinarySearch |  |  | lower(lg(n))+1 |  |

Implemented lower bound can be higher than theoretical lower bound because of used data structure. For inserting: finding the place takes O(lg(n)) time but inserting takes O(n) time on an array. If you use linkedlist inserting will take O(1) but finding the place will take O(n) time.

B-Tree, Red-Black Tree or AVL Tree have O(lg(n)) (avg or worst) time for search, insert and delete.

Insertion in a sorted list or binary search:

f(0)=0, f(1)=1

f(n) = 1 +f(n/2) = lg(n) + c n:pair

f(n) = 1+ f((n-1)/2) = lg(n+1) + c n:impair

Once you have theo lower bound that can lead you to find the structure of the algo.

Sorting: f(n)= lg(n!) => f(n) = nlg(n) + n/2 => f(n) = 2f(n/2) +n (n:partition, f(n/2): sort divided list)

f(n)= lg(n!) => f(n) = lg(n) + f(n-1) or f(n)= 1 + 2 … + lg(n-1) + lg(n) => insertion sort. But insertion sort’s complexity is higher than lg(n!) because of data structure used.

Sorting lower bounds:

Way1: Sorting n elements takes C time. Sorting 2n elements takes time of sorting 2 lists of n plus merging them: f(n) = C, f(2n) = 2C + 2n - 1

Recurence eq: f(n) = 1/2 (c_1 - 2) n + (n log(n))/(log(2)) + 1 (c_1 is an arbitrary parameter)

Way2: Sorting n elements takes C time. Inserting an element to the sorted list takes lg(n) time

f(n) = C, f(n+1) = C + lg(n)

Recurrence eq: f(n) = c_1 + (log(Γ(n)))/(log(2)) (c_1 is an arbitrary parameter) https://www4c.wolframalpha.com/Calculate/MSP/MSP9472181984da9c0583700004861bab3i6bcfee0?MSPStoreType=image/gif&s=54

Bublesort: f(n)=f(n-1) + n (recursive) => f(n)=n(n+1)/2 + c (nested loop)

**Can this be a technique to solve problems**?

1. Find the lower bound (complexity function)
2. Develop it to lead to different solutions. You can develop recursive or closed forms

Fibo: fn = fn-1 + fn-2 => fibo(n) { return n<2 ? n : fibo(n-1) + fibo(n-2) } => O(2^n)

 F_n=((1+sqrt(5))^n-(1-sqrt(5))^n)/(2^nsqrt(5)).  => O(1)

*Finding max of a list without allocating any new space? (if you don’t remember anything) you have to do (n-1)^2 comparisons. If you allocate constant (but unlimited = size of an element) space then you need n-1 comparisons. By allowing more space you get better time complexity.*

Finding an algo is an NP-Hard problem. Finding a solution is hard (exponential) and also checking the solution is hard. But it can be NP-complete thanks to model checking. A found solution can be easily verified.

**Work to do:** 1+2+…+n=? brute force search for a closed solution with funcs {mul(), add(),div()} constants{n,1,2} #ops=3

Eliminate :

* func(constant, constant)
* div(expr,0)
* mul(expr,1)
* div(mul(expr1,expr2), exprDivisor) exprDivisor<>expr1,expr2
* mul(expr1,expr2) = mul(expr2,expr1), same for add

Language:

main: expr

expr: func(expr, expr) | constant | variable

Exprs()

foreach constant yield return new Expr(c);

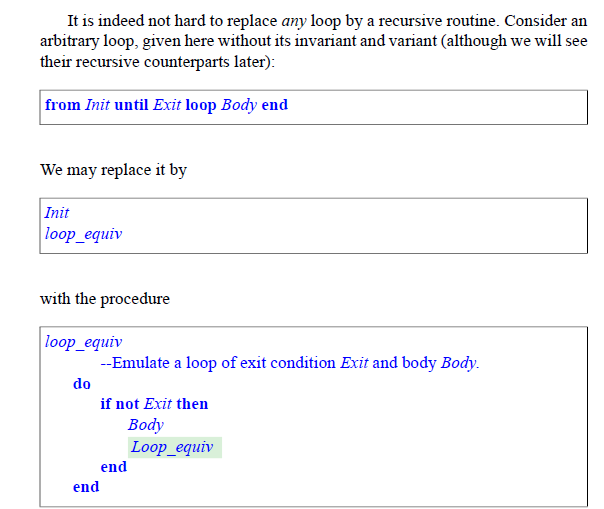
foreach variable yield return new Expr(v) ;

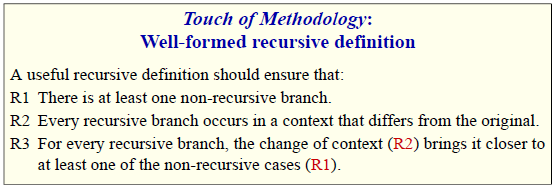
foreach func yield return new Expr(f, Exprs(),Exprs()) ;

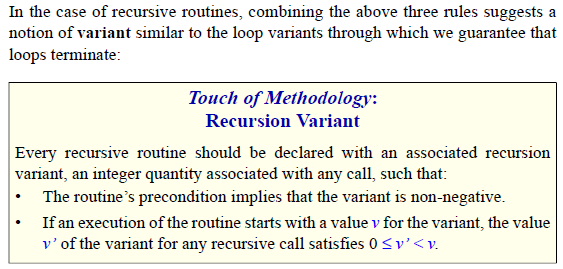
Isi gucu birak bu herifi oku: <http://jeffe.cs.illinois.edu/teaching/algorithms/>

<http://touch.ethz.ch/> Recursion chapter

**14.6 FROM LOOPS TO RECURSION**





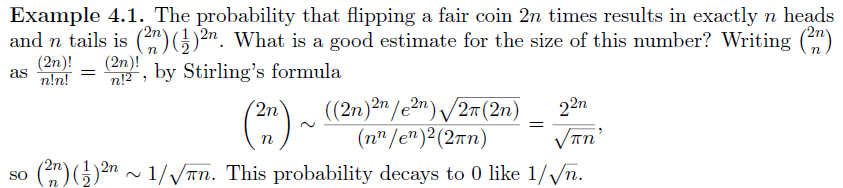


Like for other hard problems heuristics are welcomed. Read [here](https://en.wikipedia.org/wiki/Heuristic_(computer_science))

Algorithms for searching virtual spaces are used in [constraint satisfaction problem](https://en.wikipedia.org/wiki/Constraint_satisfaction_problem), where the goal is to find a set of value assignments to certain variables that will satisfy specific mathematical [equations](https://en.wikipedia.org/wiki/Equation) and [inequations](https://en.wikipedia.org/wiki/Inequation) / inequalities. They are also used when the goal is to find a variable assignment that will [maximize or minimize](https://en.wikipedia.org/wiki/Discrete_optimization) a certain function of those variables. Algorithms for these problems include the basic [brute-force search](https://en.wikipedia.org/wiki/Brute-force_search) (also called "naïve" or "uninformed" search), and a variety of [heuristics](https://en.wikipedia.org/wiki/Heuristic_function) that try to exploit partial knowledge about structure of the space, such as linear relaxation, constraint generation, and [constraint propagation](https://en.wikipedia.org/wiki/Local_consistency).

**Fitness function for sorting:** returns 0..n(n-1)/2 where O is the worst n(n-1)/2 is the best solution. That’s quadratic, can you find a nlg(n) time fitness function?

To read: Compiler optimizations, self modifying code, Algorithms in Combinatorial Geometry



**Searching algo applications**

**Sorting:**

**Multiplication:**

Peasant\* multiplication algorithm exploits the following identity: 

Karatsuba’s\* FastMultiply exploits: *ac* + *bd* − (*a* − *b*)(*c* − *d*) = *bc* + *ad*

*Can you find these identities by searching algo? bc+ad= {3 binary funcs, 4 variables, 7 operations} 3742^3=143M. You have to find first peasant’s then Karatsuba’s identities.*

\*Jeff Erickson, [All algoritms lecture notes in one file](http://jeffe.cs.illinois.edu/teaching/algorithms/)

### CPU things:

### For [ALU](https://en.wikipedia.org/wiki/Arithmetic_logic_unit) data=code. Can you modify code while execution in order to improve execution time? How about using [Goto](https://en.wikipedia.org/wiki/Goto)?

### Search[[edit](https://en.wikipedia.org/w/index.php?title=Heuristic_(computer_science)&action=edit&section=6" \o "Edit section: Search)]

Another example of heuristic making an algorithm faster occurs in certain search problems. Initially, the heuristic tries every possibility at each step, like the full-space search algorithm. But it can stop the search at any time if the current possibility is already worse than the best solution already found. In such search problems, a heuristic can be used to try good choices first so that bad paths can be eliminated early (see [alpha-beta pruning](https://en.wikipedia.org/wiki/Alpha-beta_pruning)).

### Newell and Simon: heuristic search hypothesis[[edit](https://en.wikipedia.org/w/index.php?title=Heuristic_(computer_science)&action=edit&section=7" \o "Edit section: Newell and Simon: heuristic search hypothesis)]

In their [Turing Award](https://en.wikipedia.org/wiki/Turing_Award) acceptance speech, [Allen Newell](https://en.wikipedia.org/wiki/Allen_Newell) and [Herbert A. Simon](https://en.wikipedia.org/wiki/Herbert_A._Simon) discuss the heuristic search hypothesis: a physical symbol system will repeatedly generate and modify known symbol structures until the created structure matches the solution structure. Each successive iteration depends upon the step before it, thus the heuristic search learns what avenues to pursue and which ones to disregard by measuring how close the current iteration is to the solution. Therefore, some possibilities will never be generated as they are measured to be less likely to complete the solution.

A heuristic method can accomplish its task by using search trees. However, instead of generating all possible solution branches, a heuristic selects branches more likely to produce outcomes than other branches. It is selective at each decision point, picking branches that are more likely to produce solutions.[[3]](https://en.wikipedia.org/wiki/Heuristic_(computer_science)#cite_note-3)

Read: <https://en.wikipedia.org/wiki/Mathematical_induction>

Read: <https://en.wikipedia.org/wiki/Combinatorial_proof>

Have a look at <https://www.amazon.com/Proofs-that-Really-Count-Combinatorial/dp/0883853337/ref=sr_1_1?ie=UTF8&qid=1484920106&sr=8-1&keywords=Proofs+That+Really+Count#reader_0883853337> Proves by counting is interesting

<https://en.wikipedia.org/wiki/Combinatorics>

Have a look at: <http://aggregate.ee.engr.uky.edu/MAGIC/> => a lot of low level efficient ways of computation

I think it’s enough to test 2 consecutive numbers

Wolfram alpha pattern finder

Bublesort:

2: 1

3: 1 2 1

4: 1 2 3 1 2 1

5: 1 2 3 4 1 2 3 1 2 1

Recursive; f(n): 1..n f(n-1)

Read: <https://en.wikipedia.org/wiki/DSPACE#Relation_with_other_complexity_classes>

Read heuristic search documents: A\* algorithm and swap() function finds a solution for TSP.

Learn lambda expressions: there are lists and integers. May be it’s easier to produce code. Learn about [Encoding Boolean Operations](http://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html#boolOps), [Encoding Lists](http://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html#lists), [Encoding Natural Numbers](http://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html#num), [Defining Recursive Functions](http://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html#rec) (read about other [meta programming languages](http://homepage.divms.uiowa.edu/~astump/papers/archon.pdf))

**Write how algorithms can be searched?**

Explain how to get use of lower bound, heuristics…etc

Sorting: Explain how to find buble sort. How can you search for others.

Multiplication: search for equations to find karastubas algo.

Search for others.

Read this if time: <https://arxiv.org/pdf/cs/0004001v1.pdf>

Difference between Big-O and Little-O Notation

These both describe upper bounds, Little-o is the stronger statement.

Big-O can be read as "f ∈ O(g) means that f's asymptotic growth is no faster than g's", whereas "f ∈ o(g) means that f's asymptotic growth is strictly slower than g's". It's like <= versus <.

The following are true for Big-O, but would not be true if you used little-o:

•x^2 ∈ O(x^2)

•x^2 ∈ O(x^2 + x)

•x^2 ∈ O(200 \* x^2)

The following are true for little-o:

•x^2 ∈ o(x^3)

•x^2 ∈ o(x!)

•ln(x) ∈ o(x)

Read: <http://www.leda-tutorial.org/en/official/ch02s02s03.html>

Read: <https://brohrer.github.io/blog.html> => data scince



Think about a recursive function Func() which takes itself as argument

Func(Func func){}

**References:**

**[WIL92]** Herbert S. Wilf. *Generatingfunctionology*. Academic Press, Inc, 1992

https://www.math.upenn.edu/~wilf/gfologyLinked2.pdf