Reduce and conquer

Atli FF

August 31, 2025

School of Computer Science
Reykjavík University

Divide and conquer

- Given an instance of the problem, the basic idea is to
  1. split the problem into one or more smaller subproblems
  2. solve each of these subproblems recursively
  3. combine the solutions to the subproblems into a solution of the given problem
- Some standard divide and conquer algorithms:
  - Quicksort / Mergesort
  - Karatsuba algorithm
  - Strassen algorithm
  - Many algorithms from computational geometry
    - Convex hull
    - Closest pair of points

Divide and conquer: Time complexity

```
void solve(int n) {
    if (n == 0)
        return;

    solve(n/2);
    solve(n/2);

    for (int i = 0; i < n; i++) {
        // some constant time operations
    }
}
```

- What is the time complexity of this divide and conquer algorithm?
- Usually helps to model the time complexity as a recurrence relation:
    - $T(n) = 2T(n/2) + n$

Divide and conquer: Time complexity

- But how do we solve such recurrences?
- Usually simplest to use the Master theorem when applicable
  - It gives a solution to a recurrence of the form $T(n) = aT(n/b) + f(n)$ in asymptotic terms
  - All of the divide and conquer algorithms mentioned so far have a recurrence of this form
- The Master theorem tells us that $T(n) = 2T(n/2) + n$ has asymptotic time complexity $O(n \log n)$
- You don't need to know the Master theorem for this course, but still recommended as it's very useful

Reduce and conquer

- Sometimes we're not actually dividing the problem into many subproblems, but only into one smaller subproblem
- Usually called reduce and conquer
- The most common example of this is binary search
- We will look at reduce and conquer this week, and more general divide and conquer algorithms next week

Binary search

- We have a **sorted** array of elements, and we want to check if it contains a particular element x
- Algorithm:
    1. Base case: the array is empty, return false
    2. Compare x to the element in the middle of the array
    3. If it's equal, then we found x and we return true
    4. If it's less, then x must be in the left half of the array
        4.1 Binary search the element (recursively) in the left half
    5. If it's greater, then x must be in the right half of the array
        5.1 Binary search the element (recursively) in the right half

Binary search

```
bool binary_search(const vector<int> &arr, int lo, int hi, int x) {
    if (lo > hi) {
        return false;
    }

    int m = (lo + hi) / 2;
    if (arr[m] == x) {
        return true;
    } else if (x < arr[m]) {
        return binary_search(arr, lo, m - 1, x);
    } else if (x > arr[m]) {
        return binary_search(arr, m + 1, hi, x);
    }
}

binary_search(arr, 0, arr.size() - 1, x);
```

- $T(n) = T(n/2) + 1$
- $O(\log n)$

Binary search - iterative

```
bool binary_search(const vector<int> &arr, int x) {
    int lo = 0,
        hi = arr.size() - 1;

    while (lo <= hi) {
        int m = (lo + hi) / 2;
        if (arr[m] == x) {
            return true;
        } else if (x < arr[m]) {
            hi = m - 1;
        } else if (x > arr[m]) {
            lo = m + 1;
        }
    }

    return false;
}
```

Binary search over integers

- This might be the most well known application of binary search, but it's far from being the only application
- More generally, we have a predicate $p : \{0, \ldots, n-1\} \to \{T, F\}$ which has the property that if $p(i) = T$, then $p(j) = T$ for all $j > i$
- Our goal is to find the smallest index j such that $p(j) = T$ as quickly as possible

i	0	1	$\cdots$	$j-1$	j	$j+1$	$\cdots$	$n-2$	$n-1$
$p(i)$	F	F	$\cdots$	F	T	T	$\cdots$	T	T

- We can do this in $O(\log(n) \times f)$ time, where f is the cost of evaluating the predicate p, in the same way as when we were binary searching an array

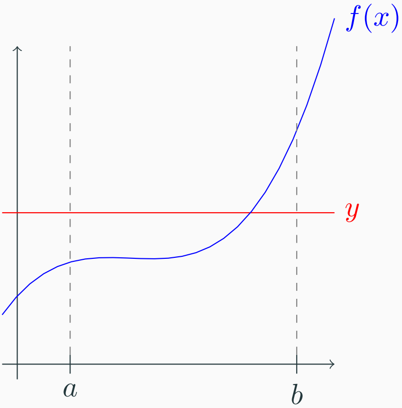
Binary search over integers

```
int lo = 0,
    hi = n - 1;

while (lo < hi) {
    int m = (lo + hi) / 2;

    if (p(m)) {
        hi = m;
    } else {
        lo = m + 1;
    }
}

if (lo == hi && p(lo)) {
    printf("lowest index is %d\n", lo);
} else {
    printf("no such index\n");
}
```

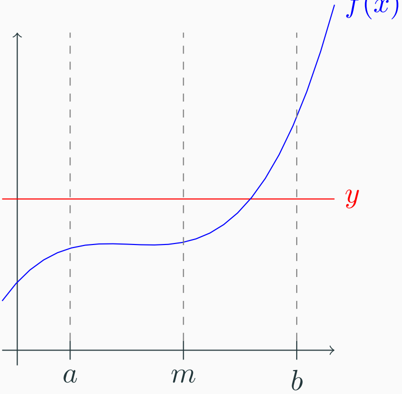
Binary search over reals

- An even more general version of binary search is over the real numbers
- We have a predicate $p : [lo, hi] \to \{T, F\}$ which has the property that if $p(i) = T$, then $p(j) = T$ for all $j > i$
- Our goal is to find the smallest real number j such that $p(j) = T$ as quickly as possible
- Since we're working with real numbers (hypothetically), our $[lo, hi]$ can be halved infinitely many times without ever becoming a single real number
- Instead it will suffice to find a real number j' that is very close to the correct answer j, say not further than $EPS = 2^{-30}$ away, we can do this in $O(\log(\frac{hi-lo}{EPS}))$ time in a similar way as when we were binary searching an array

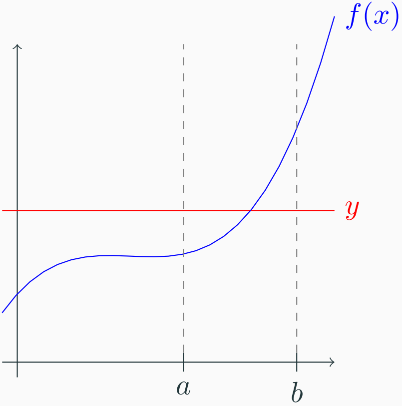
Example



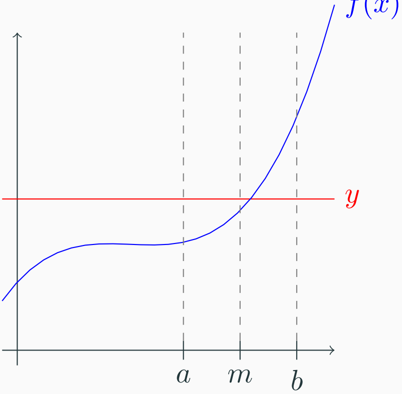
Example



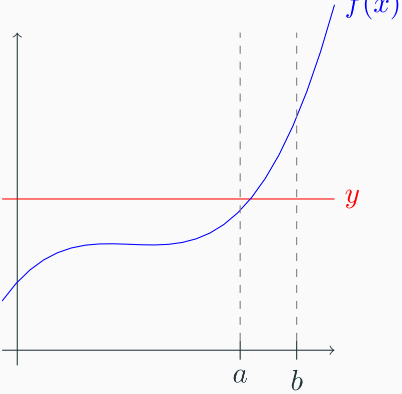
Example



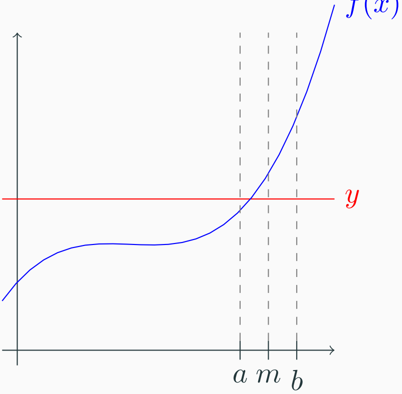
Example



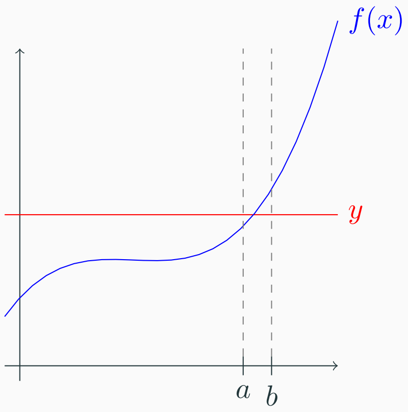
Example



Example



Example



Binary search over reals

```
double EPS = 1e-10,
       lo = -1000.0,
       hi = 1000.0;

while (hi - lo > EPS) {
    double mid = (lo + hi) / 2.0;

    if (p(mid)) {
        hi = mid;
    } else {
        lo = mid;
    }
}

printf("%0.10lf\n", lo);
```

Binary search over reals

- This has many cool numerical applications
- Find the square root of x

```
bool p(double j) {
    return j*j >= x;
}
```

- Find the root of an increasing function $f(x)$

```
bool p(double x) {
    return f(x) >= 0.0;
}
```

- This is also referred to as the Bisection method

Binary search the answer

- It may be hard to find the optimal solution directly, as we saw in the example problem
- On the other hand, it may be easy to check if some x is a solution or not
- A method of using binary search to find the minimum or maximum solution to a problem
- Only applicable when the problem has the binary search property: if i is a solution, then so are all $j > i$
- $p(i)$ checks whether i is a solution, then we simply apply binary search on p to get the minimum or maximum solution

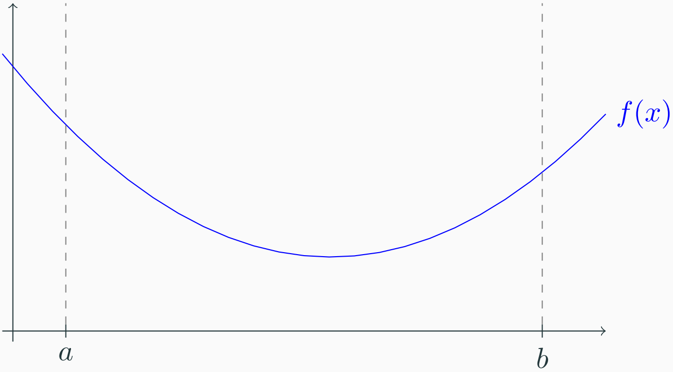
Ternary search

- Another useful and similar algorithm is ternary search
- This time we have a convex function f ($f'' \geq 0$), so it might decrease at first and then increase
- This function will have a unique minimum value that we might want to find, perhaps to minimize some cost function
- This can be done in a similar fashion to binary search

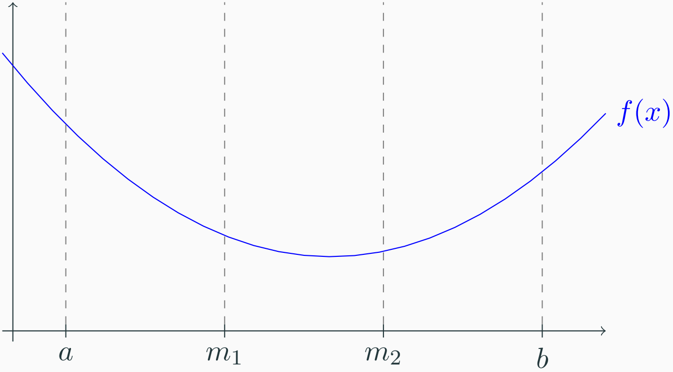
Ternary search

- We choose points m_1, m_2 in our interval $[a, b]$ so $[a, m_1]$, $[m_1, m_2]$ and $[m_2, b]$ are equally large.
- We then consider $f(m_1), f(m_2)$.
- If $f(m_1) < f(m_2)$ the minimum can not be in $[m_2, b]$ so we can discard it.
- If $f(m_2) < f(m_1)$ the minimum can not be in $[a, m_1]$ so we can discard it.
- If $f(m_1) = f(m_2)$ the minimum must be in $[m_1, m_2]$.
- This can be shown to be true using analysis, since convex functions take their maxima on the endpoints of intervals.

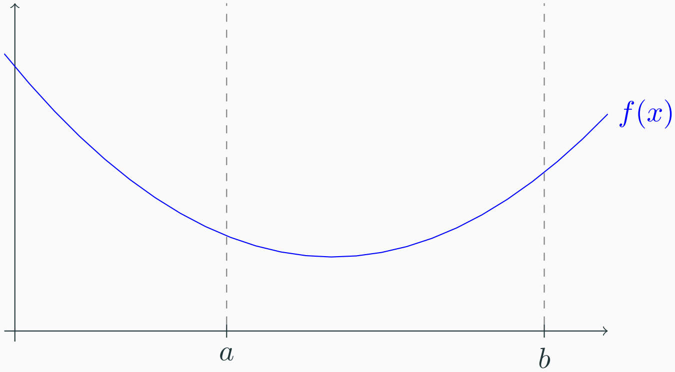
Example



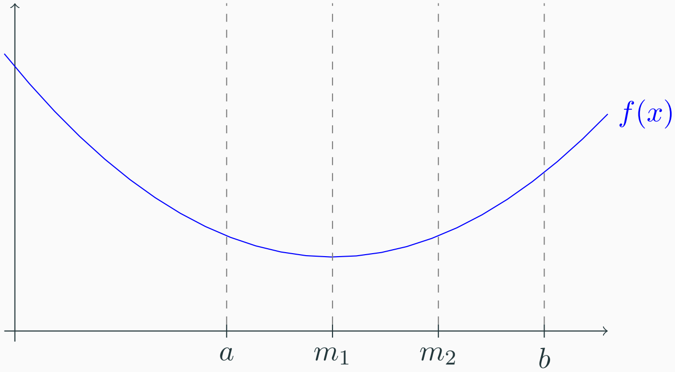
Example



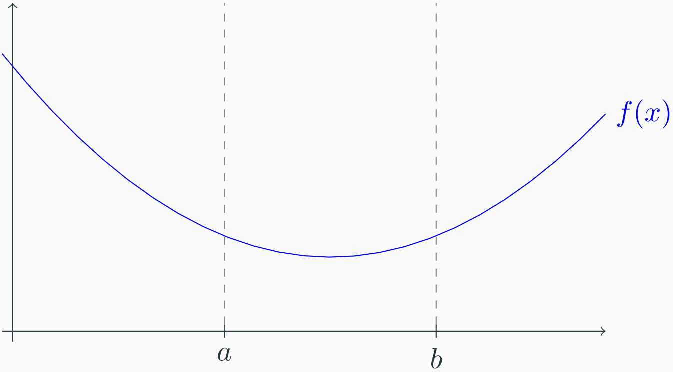
Example



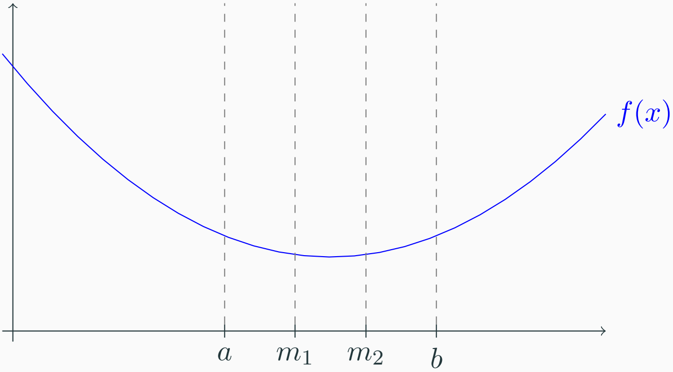
Example



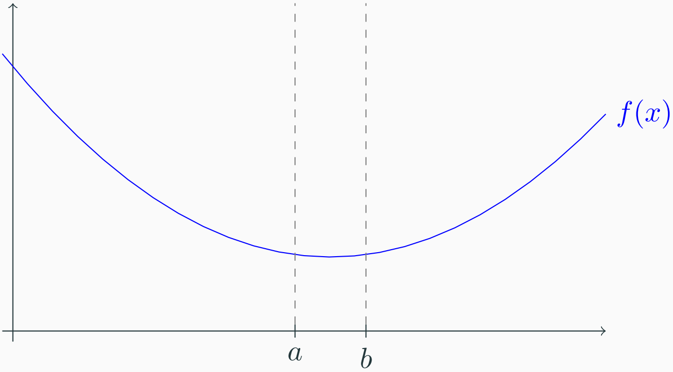
Example



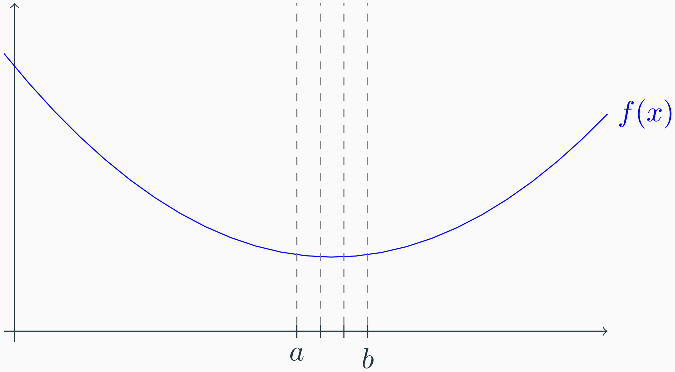
Example



Example



Example



Example