Divide & Conquer and Dynamic Programming Part 1

Atli FF

Árangursrík forritun og lausn verkefna

School of Computer Science

Reykjavík University

Divide and conquer

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 like closest point pair

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

Decrease and conquer

- Sometimes we’re not actually dividing the problem into many subproblems, but only into one smaller subproblem
- Usually called decrease and conquer
- The most common example of this is binary search

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

```cpp
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

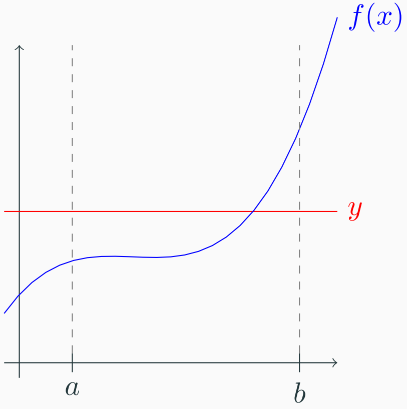
Binary search over integers

```c
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");
}
```

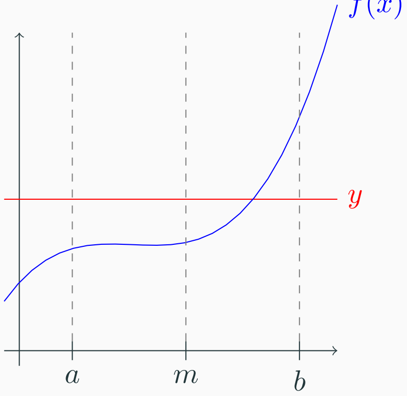
Binary search over reals

- 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

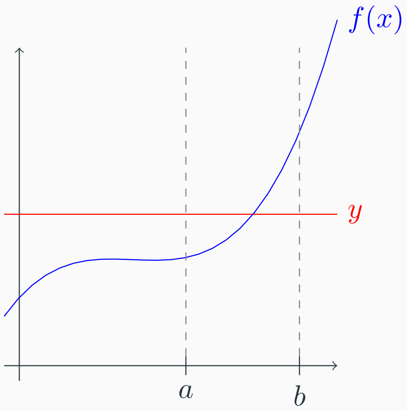
Example



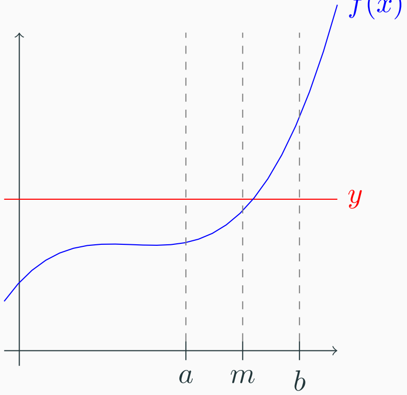
Example



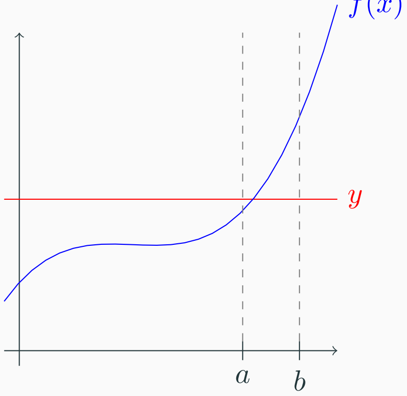
Example



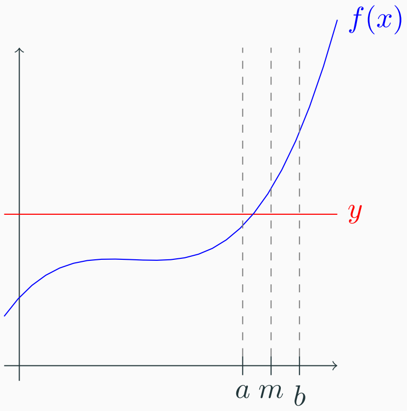
Example



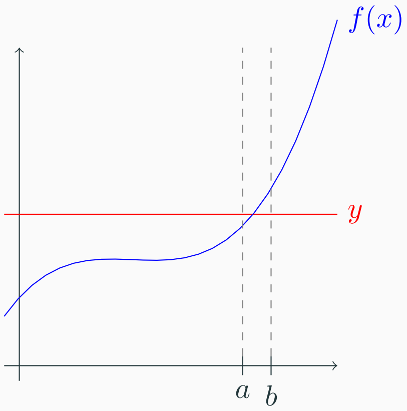
Example



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

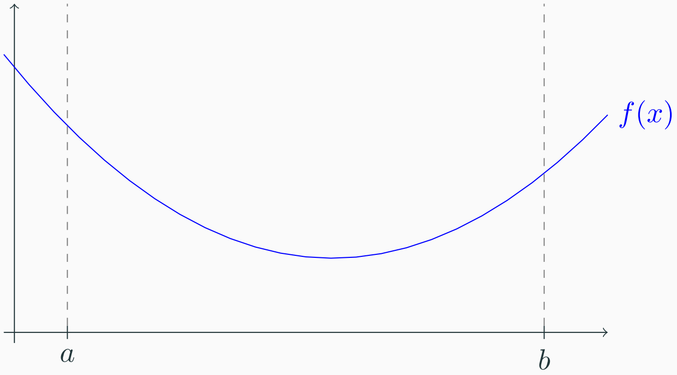
Ternary search

- 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

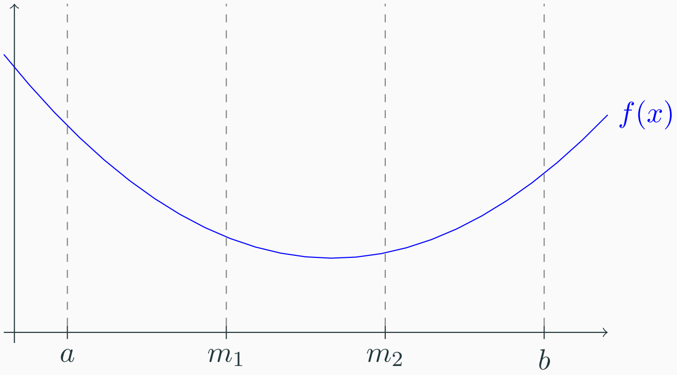
Ternary search

- 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.

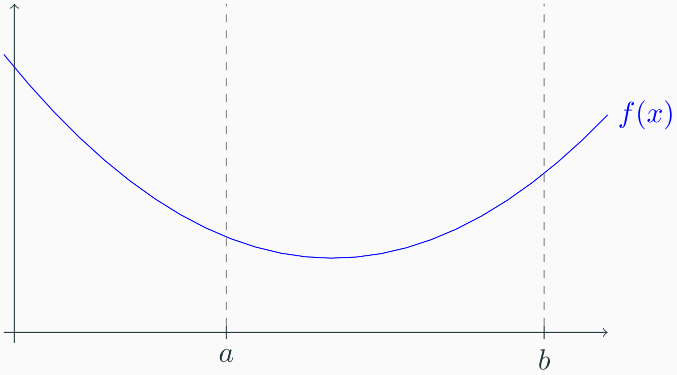
Example



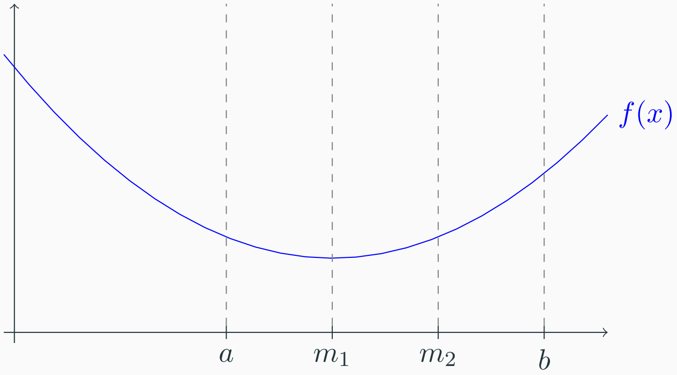
Example



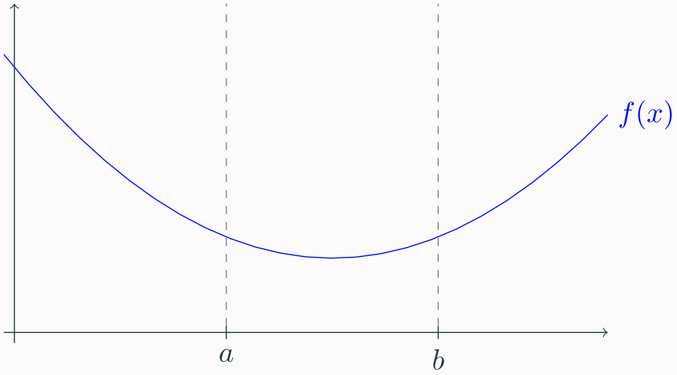
Example



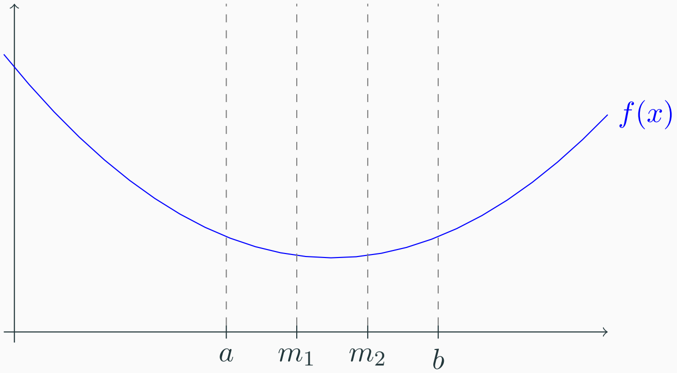
Example



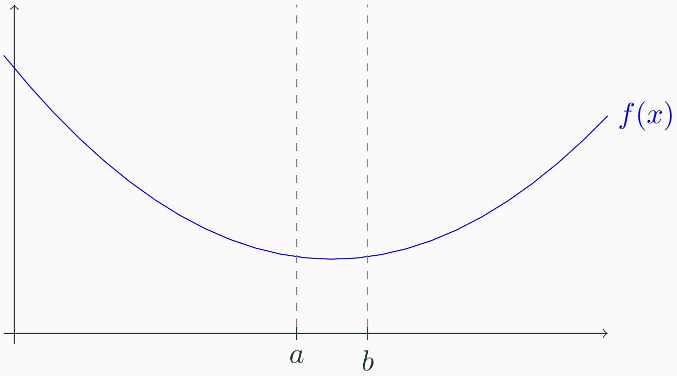
Example



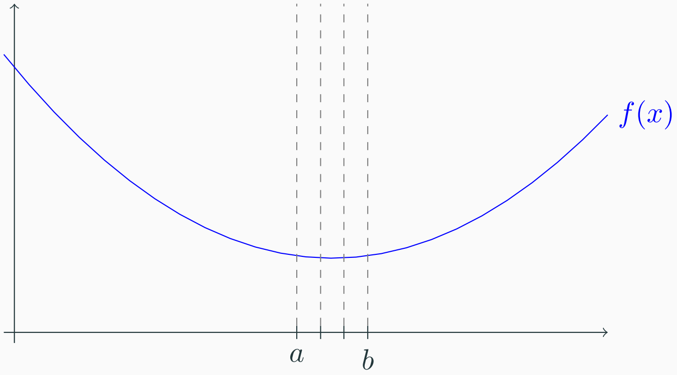
Example



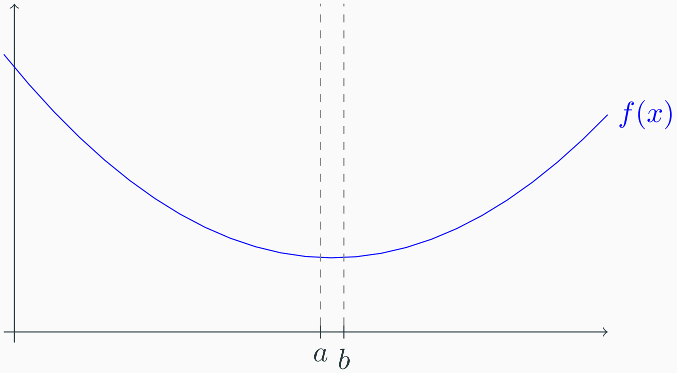
Example



Example



Example



Binary exponentiation

- We want to calculate x^n, where x, n are integers
- Assume we don't have the built-in `pow` method
- Naive method:

```
int pow(int x, int n) {
    int res = 1;
    for (int i = 0; i < n; i++) {
        res = res * x;
    }

    return res;
}
```

- This is $O(n)$, but what if we want to support large n efficiently?

Binary exponentiation

- Let's use divide and conquer

- Notice the three identities:
  - $x^0 = 1$
  - $x^n = x \times x^{n-1}$
  - $x^n = x^{n/2} \times x^{n/2}$
- Or in terms of our function:
  - $pow(x, 0) = 1$
  - $pow(x, n) = x \times pow(x, n-1)$
  - $pow(x, n) = pow(x, n/2) \times pow(x, n/2)$
- $pow(x, n/2)$ is used twice, but we only need to compute it once:
  - $pow(x, n) = pow(x, n/2)^2$

Binary exponentiation

- Let's try using these identities to compute the answer recursively

```
int pow(int x, int n) {
    if (n == 0) return 1;
    return x * pow(x, n - 1);
}
```

Binary exponentiation

- Let's try using these identities to compute the answer recursively

```
int pow(int x, int n) {
    if (n == 0) return 1;
    return x * pow(x, n - 1);
}
```

- How efficient is this?
  - $T(n) = 1 + T(n - 1)$

Binary exponentiation

- Let's try using these identities to compute the answer recursively

```
int pow(int x, int n) {
    if (n == 0) return 1;
    return x * pow(x, n - 1);
}
```

- How efficient is this?
  - $T(n) = 1 + T(n-1)$
  - $O(n)$

Binary exponentiation

- Let's try using these identities to compute the answer recursively

```
int pow(int x, int n) {
    if (n == 0) return 1;
    return x * pow(x, n - 1);
}
```

- How efficient is this?
    - $T(n) = 1 + T(n-1)$
    - $O(n)$
    - Still just as slow...

Binary exponentiation

- What about the third identity?
    - $n/2$ is not an integer when n is odd, so let's only use it when n is even

```
int pow(int x, int n) {
    if (n == 0) return 1;
    if (n % 2 != 0) return x * pow(x, n - 1);
    int st = pow(x, n/2);
    return st * st;
}
```

- How efficient is this?

Binary exponentiation

- What about the third identity?
    - $n/2$ is not an integer when n is odd, so let's only use it when n is even

```
int pow(int x, int n) {
    if (n == 0) return 1;
    if (n % 2 != 0) return x * pow(x, n - 1);
    int st = pow(x, n/2);
    return st * st;
}
```

- How efficient is this?
    - $T(n) = 1 + T(n-1)$ if n is odd
    - $T(n) = 1 + T(n/2)$ if n is even

Binary exponentiation

- What about the third identity?
  - $n/2$ is not an integer when n is odd, so let's only use it when n is even

```
int pow(int x, int n) {
    if (n == 0) return 1;
    if (n % 2 != 0) return x * pow(x, n - 1);
    int st = pow(x, n/2);
    return st * st;
}
```

- How efficient is this?
  - $T(n) = 1 + T(n - 1)$ if n is odd
  - $T(n) = 1 + T(n/2)$ if n is even
  - $T(n) = 1 + 1 + T((n - 1)/2)$ if n is odd

Binary exponentiation

- What about the third identity?
  - $n/2$ is not an integer when n is odd, so let's only use it when n is even

```
int pow(int x, int n) {
    if (n == 0) return 1;
    if (n % 2 != 0) return x * pow(x, n - 1);
    int st = pow(x, n/2);
    return st * st;
}
```

- How efficient is this?
  - $T(n) = 1 + T(n-1)$ if n is odd
  - $T(n) = 1 + T(n/2)$ if n is even
  - $T(n) = 1 + 1 + T((n-1)/2)$ if n is odd
  - $O(\log n)$

Binary exponentiation

- Notice that x doesn't have to be an integer, and $\star$ doesn't have to be integer multiplication...
- It also works for:
  - Computing x^n, where x is a floating point number and $\star$ is floating point number multiplication
  - Computing A^n, where A is a matrix and $\star$ is matrix multiplication
  - Computing $x^n \pmod{m}$, where x is an integer and $\star$ is integer multiplication modulo m
  - Computing $x \star x \star \cdots \star x$, where x is any element and $\star$ is any associative operator
- All of these can be done in $O(\log(n) \times f)$, where f is the cost of doing one application of the $\star$ operator

Fibonacci words

- Recall that the Fibonacci sequence can be defined as follows:
  - $\mathrm{fib}_1 = 1$
  - $\mathrm{fib}_2 = 1$
  - $\mathrm{fib}_n = \mathrm{fib}_{n-2} + \mathrm{fib}_{n-1}$
- We get the sequence $1, 1, 2, 3, 5, 8, 13, 21, \ldots$
- There are many generalizations of the Fibonacci sequence
- One of them is to start with other numbers, like:
  - $f_1 = 5$
  - $f_2 = 4$
  - $f_n = f_{n-2} + f_{n-1}$
- We get the sequence $5, 4, 9, 13, 22, 35, 57, \ldots$
- What if we start with something other than numbers?

Fibonacci words

- Let's try starting with a pair of strings, and let $+$ denote string concatenation:
    - $g_1 = A$
    - $g_2 = B$
    - $g_n = g_{n-2} + g_{n-1}$
- Now we get the sequence of strings:

$$A, B, AB, BAB, ABBAB, BABABBAB,$$
$$ABBABBABABBAB,$$
$$BABABBABABBABBABABBAB, \ldots$$

Fibonacci words

- How long is g_n?
  - $\mathrm{len}(g_1) = 1$
  - $\mathrm{len}(g_2) = 1$
  - $\mathrm{len}(g_n) = \mathrm{len}(g_{n-2}) + \mathrm{len}(g_{n-1})$
- Looks familiar?
- $\mathrm{len}(g_n) = \mathrm{fib}_n$
- So the strings become very large very quickly
  - $\mathrm{len}(g_{10}) = 55$
  - $\mathrm{len}(g_{100}) = 354224848179261915075$

Fibonacci words

- Task: Compute the ith character in g_n

Fibonacci words

- Task: Compute the ith character in g_n
- Simple to do in $O(\mathrm{len}(n))$, but that is extremely slow for large n

Fibonacci words

- Task: Compute the ith character in g_n
- Simple to do in $O(\mathrm{len}(n))$, but that is extremely slow for large n
- Can be done in $O(n)$ using divide and conquer

Dynamic programming

What is dynamic programming?

- A problem solving paradigm
- Similar in some respects to both divide and conquer and backtracking
- Divide and conquer recap:
    - Split the problem into *independent* subproblems
    - Solve each subproblem recursively
    - Combine the solutions to subproblems into a solution for the given problem
- Dynamic programming:
    - Split the problem into *overlapping* subproblems
    - Solve each subproblem recursively
    - Combine the solutions to subproblems into a solution for the given problem

Dynamic programming formulation

- Formulate the problem in terms of smaller versions of the problem (recursively)
- Turn this formulation into a recursive function
- Memoize the function (remember results that have been computed)

Dynamic programming formulation

```
map<problem, value> memory;

value dp(problem P) {
    if (is_base_case(P)) {
        return base_case_value(P);
    }

    if (memory.find(P) != memory.end()) {
        return memory[P];
    }

    value result = some value;
    for (problem Q in subproblems(P)) {
        result = combine(result, dp(Q));
    }

    memory[P] = result;
    return result;
}
```

The Fibonacci sequence

The first two numbers in the Fibonacci sequence are 1 and 1. All other numbers in the sequence are defined as the sum of the previous two numbers in the sequence.

- Task: Find the nth number in the Fibonacci sequence
- Let's solve this with dynamic programming
- Formulate the problem in terms of smaller versions of the problem (recursively)

$$\begin{aligned}\text{fibonacci}(1) &= 1\\ \text{fibonacci}(2) &= 1\\ \text{fibonacci}(n) &= \text{fibonacci}(n-2) + \text{fibonacci}(n-1)\end{aligned}$$

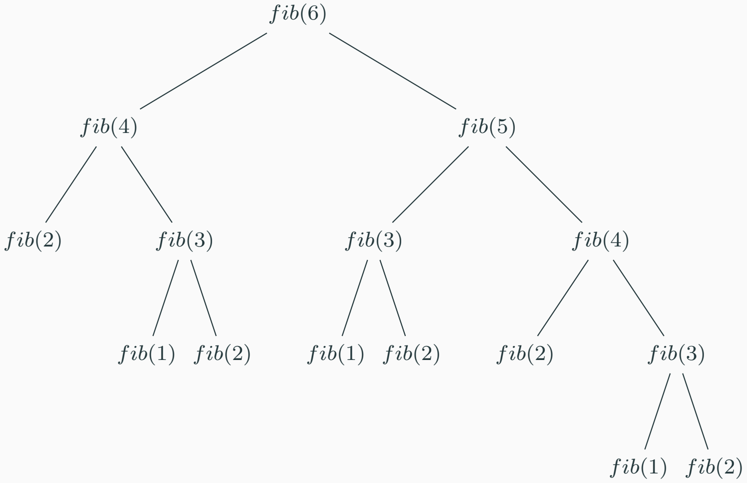
The Fibonacci sequence

2. Turn this formulation into a recursive function

```
int fibonacci(int n) {
    if (n < 2) {
        return n;
    }

    int res = fibonacci(n - 2) + fibonacci(n - 1);

    return res;
}
```

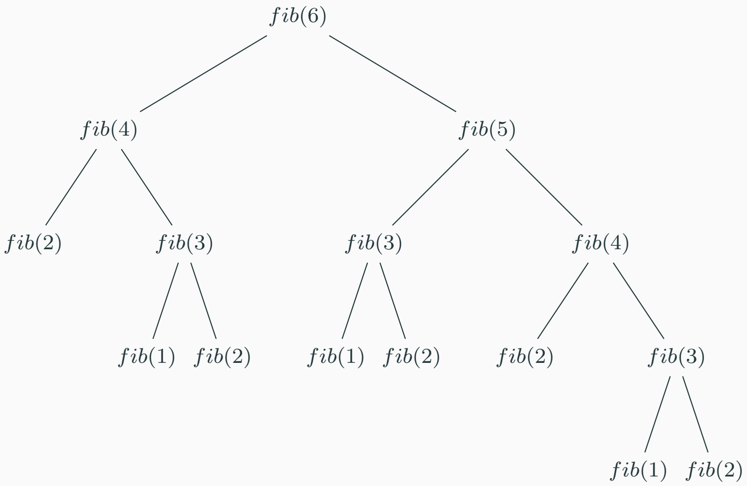
The Fibonacci sequence

- What is the time complexity of this?



The Fibonacci sequence

- What is the time complexity of this? Exponential, almost $O(2^n)$



The Fibonacci sequence

3. Memoize the function (remember results that have been computed)

```
map<int, int> mem;

int fibonacci(int n) {
    if (n <= 2) {
        return 1;
    }

    if (mem.find(n) != mem.end()) {
        return mem[n];
    }

    int res = fibonacci(n - 2) + fibonacci(n - 1);

    mem[n] = res;
    return res;
}
```

The Fibonacci sequence

```
int mem[1000];
for (int i = 0; i < 1000; i++)
    mem[i] = -1;

int fibonacci(int n) {
    if (n <= 2) {
        return 1;
    }

    if (mem[n] != -1) {
        return mem[n];
    }

    int res = fibonacci(n - 2) + fibonacci(n - 1);

    mem[n] = res;
    return res;
}
```

The Fibonacci sequence

- What is the time complexity now?
- We have n possible inputs to the function: 1, 2, $\ldots$, n.
- Each input will either:
  - be computed, and the result saved
  - be returned from memory
- Each input will be computed at most once
- Time complexity is $O(n \times f)$, where f is the time complexity of computing an input if we assume that the recursive calls are returned directly from memory ($O(1)$)
- Since we're only doing constant amount of work to compute the answer to an input, $f = O(1)$
- Total time complexity is $O(n)$

Maximum sum

- Given an array $\mathrm{arr}[0]$, $\mathrm{arr}[1]$, $\ldots$, $\mathrm{arr}[n-1]$ of integers, find the interval with the highest sum

-15	8	-2	1	0	6	-3

Maximum sum

- Given an array $\mathrm{arr}[0]$, $\mathrm{arr}[1]$, $\ldots$, $\mathrm{arr}[n-1]$ of integers, find the interval with the highest sum

-15	8	-2	1	0	6	-3

- The maximum sum of an interval in this array is 13

Maximum sum

- Given an array $\mathrm{arr}[0]$, $\mathrm{arr}[1]$, $\ldots$, $\mathrm{arr}[n-1]$ of integers, find the interval with the highest sum

-15	8	-2	1	0	6	-3

- The maximum sum of an interval in this array is 13
- But how do we solve this in general?
    - Easy to loop through all $\approx n^2$ intervals, and calculate their sums, but that is $O(n^3)$
    - We could use our static range sum trick to get this down to $O(n^2)$
    - Can we do better with dynamic programming?

Maximum sum

- First step is to formulate this recursively
- Let $\mathrm{max\_sum}(i)$ be the maximum sum interval in the range $0, \ldots, i$
- Base case: $\mathrm{max\_sum}(0) = \max(0, arr[0])$
- What about $\mathrm{max\_sum}(i)$?
- What does $\mathrm{max\_sum}(i-1)$ return?
- Is it possible to combine solutions to subproblems with smaller i into a solution for i?
- At least it's not obvious...

Maximum sum

- Let's try changing perspective
- Let $\mathrm{max\_sum}(i)$ be the maximum sum interval in the range $0, \ldots, i$, *that ends at* i
- Base case: $\mathrm{max\_sum}(0) = arr[0]$
- $\mathrm{max\_sum}(i) = \max(arr[i], arr[i] + \mathrm{max\_sum}(i-1))$
- Then the answer is just $\max_{0 \leq i < n} \{ \mathrm{max\_sum}(i) \}$

Maximum sum

- Next step is to turn this into a function

```
int arr[1000];

int max_sum(int i) {
    if (i == 0) {
        return arr[i];
    }

    int res = max(arr[i], arr[i] + max_sum(i - 1));

    return res;
}
```

Maximum sum

- Final step is to memoize the function

```
int arr[1000];
int mem[1000];
bool comp[1000];
memset(comp, 0, sizeof(comp));

int max_sum(int i) {
    if (i == 0) {
        return arr[i];
    }
    if (comp[i]) {
        return mem[i];
    }

    int res = max(arr[i], arr[i] + max_sum(i - 1));

    mem[i] = res;
    comp[i] = true;
    return res;
}
```

Maximum sum

- Then the answer is just the maximum over all interval ends

```
int maximum = 0;
for (int i = 0; i < n; i++) {
    maximum = max(maximum, max_sum(i));
}

printf("%d\n", maximum);
```

Maximum sum

- If you want to find the maximum sum interval in multiple arrays, remember to clear the memory in between

Maximum sum

- What about time complexity?
- There are n possible inputs to the function
- Each input is processed in $O(1)$ time, assuming recursive calls are $O(1)$
- Time complexity is $O(n)$

Coin change

- Given an array of coin denominations d_0, d_1, ..., d_{n-1}, and some amount x: What is minimum number of coins needed to represent the value x?
- Remember the greedy algorithm for Coin change?
- It didn't always give the optimal solution, and sometimes it didn't even give a solution at all...
- What about dynamic programming?

Coin change

- First step: formulate the problem recursively

- Let $\mathrm{opt}(i, x)$ denote the minimum number of coins needed to represent the value x if we're only allowed to use coin denominations $d_0, \ldots, d_i$

- Base case: $\mathrm{opt}(i, x) = \infty$ if $x < 0$
- Base case: $\mathrm{opt}(i, 0) = 0$
- Base case: $\mathrm{opt}(-1, x) = \infty$

- $\mathrm{opt}(i, x) = \min \begin{cases} 1 + \mathrm{opt}(i, x - d_i) \\ \mathrm{opt}(i - 1, x) \end{cases}$

Coin change

```
int INF = 100000;
int d[10];

int opt(int i, int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (i == -1) return INF;

    int res = INF;
    res = min(res, 1 + opt(i, x - d[i]));
    res = min(res, opt(i - 1, x));

    return res;
}
```

Coin change

```
int INF = 100000;
int d[10];
int mem[10][10000];
memset(mem, -1, sizeof(mem));

int opt(int i, int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (i == -1) return INF;

    if (mem[i][x] != -1) return mem[i][x];

    int res = INF;
    res = min(res, 1 + opt(i, x - d[i]));
    res = min(res, opt(i - 1, x));

    mem[i][x] = res;
    return res;
}
```

Coin change

- Time complexity?
- Number of possible inputs are $n \times x$
- Each input will be processed in $O(1)$ time, assuming recursive calls are constant
- Total time complexity is $O(n \times x)$

Coin change

- How do we know which coins the optimal solution used?
- We can store backpointers, or some extra information, to trace backwards through the states
- See example...

Longest increasing subsequence

- Given an array $a[0]$, $a[1]$, $\ldots$, $a[n-1]$ of integers, what is the length of the longest increasing subsequence?
- First, what is a subsequence?
- If we delete zero or more elements from a, then we have a subsequence of a
- Example: $a = [5, 1, 8, 1, 9, 2]$
- $[5, 8, 9]$ is a subsequence
- $[1, 1]$ is a subsequence
- $[5, 1, 8, 1, 9, 2]$ is a subsequence
- $[]$ is a subsequence
- $[8, 5]$ is **not** a subsequence

Longest increasing subsequence

- Given an array $a[0]$, $a[1]$, $\ldots$, $a[n-1]$ of integers, what is the length of the longest increasing subsequence?
- An increasing subsequence of a is a subsequence of a such that the elements are in (strictly) increasing order
- $[5, 8, 9]$ and $[1, 8, 9]$ are the longest increasing subsequences of $a = [5, 1, 8, 1, 9, 2]$
- How do we compute the length of the longest increasing subsequence?
- There are 2^n subsequences, so we can go through all of them
- That would result in an $O(n2^n)$ algorithm, which can only handle $n \leq 23$
- What about dynamic programming?

Longest increasing subsequence

- Let $\mathrm{lis}(i)$ denote the length of the longest increasing subsequence of the array $a[0]$, …, $a[i]$

- Base case: $\mathrm{lis}(0) = 1$
- What about $\mathrm{lis}(i)$?

- We have the same issue as in the maximum sum problem, so let's try changing perspective

Longest increasing subsequence

- Let $\operatorname{lis}(i)$ denote the length of the longest increasing subsequence of the array $a[0]$, …, $a[i]$, *that ends at* i

- Base case: we don't need one
- $\operatorname{lis}(i) = \max(1, \max_{j<i \text{ s.t. } a[j]<a[i]}\{1 + \operatorname{lis}(j)\})$

Longest increasing subsequence

```
int a[1000];
int mem[1000];
memset(mem, -1, sizeof(mem));

int lis(int i) {
    if (mem[i] != -1) {
        return mem[i];
    }

    int res = 1;
    for (int j = 0; j < i; j++) {
        if (a[j] < a[i]) {
            res = max(res, 1 + lis(j));
        }
    }

    mem[i] = res;
    return res;
}
```

Longest increasing subsequence

- And then the longest increasing subsequence can be found by checking all endpoints:

```
int mx = 0;
for (int i = 0; i < n; i++) {
    mx = max(mx, lis(i));
}

printf("%d\n", mx);
```

Longest increasing subsequence

- Time complexity?

- There are n possible inputs
- Each input is computed in $O(n)$ time, assuming recursive calls are $O(1)$
- Total time complexity is $O(n^2)$

- This will be fast enough for $n \leq 10\ 000$, much better than the brute force method!
- (It can be done faster ($n \log(n)$) with dynamic programming optimizations, but that's for later)

Longest common subsequence

- Given two strings (or arrays of integers) $a[0], \ldots, a[n-1]$ and $b[0], \ldots, b[m-1]$, find the length of the longest subsequence that they have in common.

- $a = \texttt{"b}\underline{\texttt{an}}\texttt{an}\underline{\texttt{inn}}\texttt{"}$
- $b = \texttt{"k}\underline{\texttt{anin}}\texttt{a}\underline{\texttt{n}}\texttt{"}$
- The longest common subsequence of a and b, `"aninn"`, has length 5

Longest common subsequence

- Let $\operatorname{lcs}(i, j)$ be the length of the longest common subsequence of the strings $a[0]$, ..., $a[i]$ and $b[0]$, ..., $b[j]$
- Base case: $\operatorname{lcs}(-1, j) = 0$
- Base case: $\operatorname{lcs}(i, -1) = 0$
- $\operatorname{lcs}(i, j) = \max \begin{cases} \operatorname{lcs}(i, j-1) \\ \operatorname{lcs}(i-1, j) \\ 1 + \operatorname{lcs}(i-1, j-1) & \text{if } a[i] = b[j] \end{cases}$

Longest common subsequence

```
string a = "bananinn",
       b = "kaninan";
int mem[1000][1000];
memset(mem, -1, sizeof(mem));

int lcs(int i, int j) {
    if (i == -1 || j == -1) {
        return 0;
    }
    if (mem[i][j] != -1) {
        return mem[i][j];
    }

    int res = 0;
    res = max(res, lcs(i, j - 1));
    res = max(res, lcs(i - 1, j));

    if (a[i] == b[j]) {
        res = max(res, 1 + lcs(i - 1, j - 1));
    }

    mem[i][j] = res;
```

Longest common subsequence

- Time complexity?

- There are $n \times m$ possible inputs
- Each input is processed in $O(1)$, assuming recursive calls are $O(1)$
- Total time complexity is $O(n \times m)$

Top-down vs. bottom-up

- What we have been doing so far is usually called top-down dynamic programming
- I.e. you start with the main problem (the top) and split it into smaller problems recursively (down)
- In some cases it can be better to do things bottom-up, which is pretty much just the reverse order
- Consider for example the fibonacci numbers. Then we'd start at the base cases and count up

Top-down vs. bottom-up

- Bottom-up is generally faster, but it has the issue that we have to make sure we iterate through the sub problems in the right order, since recursion doesn't automatically handle that for us
- Then why would we use bottom-up?
- Sometimes knowing in what order we go through the states can be useful, let's take an example

Decelerating jump

- Suppose we have n squares in a row, each containing a value a_i.
- We start at the first cell and want to jump through the cells.
- If we land on cell i we get a_i points and want to maximize our points.
- We can jump to any cell in front of us, but our jump can never go further than our last one.
- $n \leq 3000$, $-10^9 \leq a_i \leq 10^9$

Solution function

- Let us then find a recursive function describing the answer.
- Let $f(i, j)$ be the maximum score one can get starting at i and using jumps of at most length j. Then

$$f(i,j) = \begin{cases} a_n & \text{if } i = n \\ \max\limits_{1 \le k \le \min(j, n-i)} f(i+k, k) + a_i & \text{otherwise} \end{cases}$$

- Then we have n^2 states and it takes linear time to calculate each one. Thus a top-down solution would run in $\mathcal{O}(n^3)$, which is too slow.

Optimization

- What about bottom up?
- Each $f(i, j)$ only depends on values of f with greater i and lesser j, so we can calculate them in increasing order of j and decreasing order of i.

First implementation

```
#define INF (1LL << 60)
int main() {
    ll n; cin >> n;
    ll d[n][n], a[n];
    for(int i = 0; i < n; ++i) cin >> a[i];
    for(int i = 0; I < n; ++i) for(int j = 0; j < n; ++j) d[i][j] = -INF;
    d[n - 1][1] = a[n - 1];
    for(int i = n - 2; i >= 0; --i) d[i][1] = d[i + 1][1] + a[i];
    for(int i = 0; i < n; ++i) d[n - 1][i] = a[n - 1];
    for(int j = 2; j < n; ++j) for(int i = n - 2; i >= 0; --i)
        for(int k = 1; k < min(j + 1, n - i); k++)
            d[i][j] = max(d[i][j], d[i + k][k] + a[i]);
    cout << d[0][n - 1] << '\n';
}
```

Optimized?

- This is still $\mathcal{O}(n^3)$, so still not good enough.
- We note that when we calculate $f(i, j)$ we use the maximum value along the diagonal $f(i+1, 1), f(i+2, 2), \ldots, f(i+k, k)$.
- So we just add a prefix array for the diagonals to calculate those values in $\mathcal{O}(1)$.
- This will make the solution $\mathcal{O}(n^2)$, which is good enough
- This particular case can be done top-down as well, but as we do more complex optimizations of this kind, that may no longer hold

Fast implementation

```
#define INF (1LL << 60)
int main() {
    ll n; cin >> n;
    ll d[n][n], e[n], a[n];
    for(int i = 0; i < n; ++i) cin >> a[i];
    for(int i = 0; I < n; ++i) for(int j = 0; j < n; ++j) d[i][j] = -INF;
    d[n - 1][1] = e[n - 1] = a[n - 1];
    for(int i = n - 2; i >= 0; --i) e[i] = d[i][1] = d[i + 1][1] + a[i];
    for(int i = 0; i < n; ++i) d[n - 1][i] = a[n - 1];
    for(int j = 2; j < n; ++j) {
        e[n - j] = max(e[n - j], d[n - 1][j] = a[n - 1]);
        for(int i = n - 2; i >= 0; --i) {
            d[i][j] = e[i + 1] + a[i];
            if(i >= j - 1) e[i - j + 1] = max(e[i - j + 1], d[i][j]);
        }
    }
    cout << d[0][n - 1] << '\n';
}
```