Divide and conquer, Dynamic programming

Arnar Bjarni Arnarson & Atli FF

September 8, 2025

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
    - 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

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

Fibonacci words

- Task: Compute the ith character in g_n

Fibonacci words

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

Fibonacci words

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

Mergesort

- Input is a sequence of n elements $A_1, A_2, \ldots, A_n$.
- Base case when $n = 1$, just return sequence
- Otherwise, split into two (almost) equal halves
- Recursively sort sequences $A_1, \ldots A_{\lfloor \frac{n}{2} \rfloor}$ and $A_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, A_n$.
- Create new sequence by interleaving the two, always picking the lower front value.
- Mergesort is an $\mathcal{O}(n \log n)$ sorting algorithm

Inversions

- An inversion is a pair of out of order elements.
- Consider the permutation $6, 2, 3, 1, 5, 4$
- $(6, 2)$ form an inversion
- $(2, 5)$ do not form an inversion
- There are $5 + 1 + 1 + 0 + 1 + 0 = 8$ inversions in the permutation.
- Problem: Given permutation of size $n \leq 10^6$, compute number of inversions.

Counting inversions

- Recall mergesort
- Can we modify it to count inversions?

Counting inversions

- Recall mergesort
- Can we modify it to count inversions?
- Assume it does count inversions.
- Mergesorting left half gives us inversions there and same for right half.
- Need to compute number of inversions with one element in left half and the other in right half.

Counting inversions

- Recall mergesort
- Can we modify it to count inversions?
- Assume it does count inversions.
- Mergesorting left half gives us inversions there and same for right half.
- Need to compute number of inversions with one element in left half and the other in right half.
- When picking element from right half, add number of elements remaining in left half.
- Since sequences are sorted, we know everything remaining in left half is larger than the picked element from right half.