Problem solving paradigms

Bjarki Ágúst Guðmundsson
Tómas Ken Magnússon

School of Computer Science
Reykjavík University

Árangursrík forritun og lausn verkefna

Today we're going to cover

- Problem solving paradigms
- Complete search
- Backtracking
- Divide and conquer

Example problem

- Problem C from NWERC 2006: Pie

Problem solving paradigms

- What is a problem solving paradigm?
- A method to construct a solution to a specific type of problem
- Today and in later lectures we will study common problem solving paradigms

Complete search

- We have a finite set of objects
- We want to find an element in that set which satisfies some constraints
  - or find **all** elements in that set which satisfy some constraints
- Simple! Just go through all elements in the set, and for each of them check if they satisfy the constraints
- Of course it's not going to be very efficient...
- But remember, we always want the simplest solution that runs in time
- Complete search should be the first problem solving paradigm you think about when you're trying to solve a problem

Example problem: Vito's family

- http://uva.onlinejudge.org/external/100/10041.html

Complete search

- What if the search space is more complex?
  - All permutations of n items
  - All subsets of n items
  - All ways to put n queens on an $n \times n$ chessboard without any queen attacking any other queen
- How are we supposed to iterate through the search space?
- Let's take a better look at these examples

Iterating through permutations

- Already implemented in many standard libraries:
  - `next_permutation` in C++
  - `itertools.permutations` in Python

```
int n = 5;
vector<int> perm(n);
for (int i = 0; i < n; i++) perm[i] = i + 1;

do {
    for (int i = 0; i < n; i++) {
        printf("%d ", perm[i]);
    }
    printf("\n");

} while (next_permutation(perm.begin(), perm.end()));
```

Iterating through permutations

- Even simpler in Python...

- Remember that there are $n!$ permutations of length n, so usually you can only go through all permutations if $n \leq 11$
  - Otherwise you need to find a more clever approach than complete search

Iterating through subsets

- Remember the bit representation of subsets?
- Each integer from 0 to $2^n - 1$ represents a different subset of the set $\{1, 2, \ldots, n\}$
- Just iterate through the integers

```
int n = 5;
for (int subset = 0; subset < (1 << n); subset++) {
    for (int i = 0; i < n; i++) {
        if ((subset & (1 << i)) != 0) {
            printf("%d ", i+1);
        }
    }
    printf("\n");
}
```

Iterating through subsets

- Similar in Python
- Remember that there are 2^n permutations of length n, so usually you can only go through all permutations if $n \leq 25$
  - Otherwise you need to find a more clever approach than complete search

Backtracking

- We’ve seen two ways to go through a complex search space, but both of the solutions were rather specific
- Would be nice to have a more general “framework”
- Backtracking!

Backtracking

- Define states
  - We have one initial “empty” state
  - Some states are partial
  - Some states are complete
- Define transitions from a state to possible next states
- Basic idea:
  1. Start with the empty state
  2. Use recursion to traverse all states by going through the transitions
  3. If the current state is invalid, then stop exploring this branch
  4. Process all complete states (these are the states we’re looking for)

Backtracking

- General solution form:

```
state S;

void generate() {
    if (!is_valid(S))
        return;

    if (is_complete(S))
        print(S);

    foreach (possible next move P) {
        apply move P;
        generate();
        undo move P;
    }
}

S = empty state;
generate();
```

Generating all subsets

- Also simple to do with backtracking:

```
const int n = 5;
bool pick[n];

void generate(int at) {
    if (at == n) {
        for (int i = 0; i < n; i++) {
            if (pick[i]) {
                printf("%d ", i+1);
            }
        }
        printf("\n");
    } else {

        // either pick element no. at
        pick[at] = true;
        generate(at + 1);

        // or don't pick element no. at
        pick[at] = false;
        generate(at + 1);
    }
}

generate(0);
```

Generating all permutations

- Also simple to do with backtracking:

```
const int n = 5;
int perm[n];
bool used[n];

void generate(int at) {
    if (at == n) {
        for (int i = 0; i < n; i++) {
            printf("%d ", perm[i]+1);
        }
        printf("\n");
    } else {

        // decide what the at-th element should be
        for (int i = 0; i < n; i++) {
            if (!used[i]) {
                used[i] = true;
                perm[at] = i;

                generate(at + 1);

                // remember to undo the move:
                used[i] = false;
            }
        }
    }
}

memset(used, 0, n);
generate(0);
```

n queens

- Given n queens and an $n \times n$ chessboard, find all ways to put the n queens on the chessboard such that no queen can attack any other queen
- This is a very specific set we want to iterate through, so we probably won't find this in the standard library
- We could use our bit trick to iterate through all subsets of the $n \times n$ cells of size n, but that would be very slow
- Let's use backtracking

n queens

- Go through the cells in increasing order
- Either put a queen on that cell or not (transition)
- Don't put down a queen if she's able to attack another queen already on the table

```
const int n = 8;
bool has_queen[n][n];
int queens_left = n;

// generate function

memset(has_queen, 0, sizeof(has_queen));
generate(0, 0);
```

n queens

```
void generate(int x, int y) {
    if (y == n) {
        generate(x+1, 0);
    } else if (x == n) {
        if (queens_left == 0) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    printf("%c", has_queen[i][j] ? 'Q' : '.');
                }
                printf("\n");
            }
        }
    } else {

        if (queens_left > 0 and no queen can attack cell (x,y)) {
            // try putting a queen on this cell
            has_queen[x][y] = true;
            queens_left--;

            generate(x, y+1);

            // undo the move
            has_queen[x][y] = false;
            queens_left++;
        }

        // try leaving this cell empty
        generate(x, y+1);
    }
}
```

Example problem: The Hamming Distance Problem

- http://uva.onlinejudge.org/external/7/729.html

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

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

```
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 $p : \{0, \ldots, n-1\} \rightarrow \{T, F\}$ 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 integers

- Find the index of *x* in the sorted array *arr*

```
bool p(int i) {
    return arr[i] >= x;
}
```

- Later we'll see how to use this in other ways

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

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

Example problem

- Problem C from NWERC 2006: Pie

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

Other types of divide and conquer

- Binary search is very useful, can be used to construct simple and efficient solutions to problems
- But binary search is only one example of divide and conquer
- Let's explore two more examples

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
  - Since $n - 1$ is even when n is odd:
  - $T(n) = 1 + 1 + T((n - 1)/2)$ 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
  - Since $n - 1$ is even when n is odd:
  - $T(n) = 1 + 1 + T((n-1)/2)$ if n is odd
  - $O(\log n)$
  - Fast!

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 a matrix 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
  - $\mathrm{len}(g_{10}) = 55$
  - $\mathrm{len}(g_{100}) = 354224848179261915075$
  - $\mathrm{len}(g_{1000}) =$

    $434665576869374564356885276750406258025646605173717$
    $804024817290895365554179490518904038798400792551692$
    $959225930803226347752096896232398733224711616429964$
    $409065331879382989696499285160037044761377951668492$
    28875

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