Divide and Conquer

Pham Quang Dung and Do Phan Thuan

Computer Science Department, SoICT,
Hanoi University of Science and Technology.

February 23, 2017

Divide and Conquer

D&C is a problem-solving paradigm in which a problem is made *simpler* by ‘dividing’ it into smaller parts and then conquering each part.

There are usually 3 main steps:

1. DIVIDE: split the problem into one or more smaller subproblems - usually by half or nearly half
2. CONQUER: solve each of these subproblems recursively - which are now easier
3. COMBINE: combine the solutions to the subproblems into a solution of the given problem

Standard divide and conquer algorithms

- Quicksort
- Mergesort
- Karatsuba algorithm
- Strassen algorithm
- Many algorithms from computational geometry
    - Convex hull
    - Closest pair of points

Applications of D&C

- Solving difficult problems: breaking the problem into sub-problems, solving the trivial cases and combining sub-problems to the original problem
- Parallelism: naturally adapted for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors
- Memory access: naturally tend to make efficient use of memory caches. Once a sub-problem is small enough, it and all its sub-problems can, in principle, be solved within the cache, without accessing the slower main memory
- Roundoff control: may yield more accurate results than a superficially equivalent iterative method with rounded arithmetic. For example, one can add N numbers either by a simple loop that adds each datum to a single variable, or by a D&C algorithm called pairwise summation that breaks the data set into two halves, recursively computes the sum of each half, and then adds the two sums. While the second method performs the same number of additions as the first, and pays the overhead of the recursive calls, it is usually more accurate

Analysis of D&C

- Described by recursive equation
- Suppose $T(n)$ is the running time on a problem of size n
- $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq n_c \\ aT(n/b) + D(n) + C(n) & \text{if } n \geq n_c, \end{cases}$

  where

  a: number of subproblems
  n/b: size of each subproblem
  $D(n)$: cost of divide operation
  $C(n)$: cost of combination operation

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:
  - $T(n) = 2T(n/2) + n$

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 $O(n \log n)$
- The recurrence tree method is also very useful to solve such recurrences.

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

Merge Sort

- **Divide:** divide the n-element sequence into two subproblems of $n/2$ elements each
- **Conquer:** sort the two subsequences recursively using merge sort. If the length of a sequence is 1, do nothing since it is already in order
- **Combine** merge the two sorted subsequences to produce the sorted answer

Merge Sort – Merge Function

- Merge is the key operation in merge sort
- Suppose the (sub)sequence(s) are stored in the array A. Moreover, $A[p \dots q]$ and $A[q+1 \dots r]$ are two sorted subsequences.
- MERGE(A,p,q,r) will merge the two subsequences into sorted sequence $A[p \dots r]$
- MERGE(A,p,q,r) takes $\Theta(r-p+1)$

```
MERGE_SORT(A,p,r){
    if (p<r) {
        q=(p+r)/2

        MERGE_SORT(A,p,q)
        MERGE_SORT(A,q+1,r)
        MERGE(A,p,q,r)}
    }
```

Call to MERGE-SORT(A,1,n) (suppose n=length(A))

Analysis of Merge Sort

- **Divide:** $D(n) = \Theta(1)$
- **Conquer:** $a = 2, b = 2$, so $2T(n/2)$
- **Combine** $C(n) = \Theta(n)$
- $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$
- $T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$
- $T(n) = \mathcal{O}(n \log n)$ by Recursive Tree or Master Theorem

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
     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
     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 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

Practicing problem

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:
  - $\text{fib}_1 = 1$
  - $\text{fib}_2 = 1$
  - $\text{fib}_n = \text{fib}_{n-2} + \text{fib}_{n-1}$
- 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
  - $\text{len}(g_{10}) = 55$
  - $\text{len}(g_{100}) = 354224848179261915075$
  - $\text{len}(g_{1000}) =$

    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

Practicing problem

Fibonacci Words