2. Divide and conquer

Angel Napa

April 17, 2023

Resumen

Divide and Conquer

Mergesort

Decrease and Conquer

Binary search

Divide and Conquer

Divide and conquer

- The basic idea to a "divide and conquer" algorithm is:
  1. **Divide:** split the problem in more than one smaller subproblems
  2. **Conquer:** solve each of the subproblems recursively
  3. **Merge:** Combine the solutions of these subproblems in the original problem
- Some divide and conquer algorithms:
  1. Quicksort
  2. Mergesort
  3. Karatsuba algorithm
  4. Convex Hull
  5. etc.

Time Complexity

- $f(n) = \#$ operations in the program

Time Complexity

- $f(n) = \#$ operations in the program
- Common Notation: $f(n) = O(g(n))$

Time Complexity

- $f(n) = \#$ operations in the program
- Common Notation: $f(n) = O(g(n))$
- For n sufficiently large $f(n) \leq C \cdot g(n)$, for some $C > 0$

Time Complexity

- $f(n) = \#$ operations in the program
- Common Notation: $f(n) = O(g(n))$
- For n sufficiently large $f(n) \leq C \cdot g(n)$, for some $C > 0$
- $\alpha < \beta \rightarrow n^{\alpha} << n^{\beta}$

Time Complexity

- $f(n) = \#$ operations in the program
- Common Notation: $f(n) = O(g(n))$
- For n sufficiently large $f(n) \leq C \cdot g(n)$, for some $C > 0$
- $\alpha < \beta \rightarrow n^{\alpha} << n^{\beta}$
- $a > 1 \rightarrow \log_a(n) << n^{\alpha} << a^n$

Time Complexity

$$T(n) = 2 * T(n/2) + n$$

- How do we know this time of complexity?

Time Complexity

$$T(n) = 2 * T(n/2) + n$$

- How do we know this time of complexity?
- Check **the Master Theorem**:

Time Complexity

$$T(n) = 2 * T(n/2) + n$$

- How do we know this time of complexity?
- Check **the Master Theorem**:
- It helps to know the complexity of algoritthms where $T(n) = aT(n/b) + f(n)$

Time Complexity

$$T(n) = 2 * T(n/2) + n$$

- How do we know this time of complexity?
- Check **the Master Theorem**:
- It helps to know the complexity of algoritthms where $T(n) = aT(n/b) + f(n)$
- Thanks to this we can conclude that the time complexity is $O(nlog(n))$

Mergesort

Mergesort

Given an $array[a...b]$:

1. if $a = b$ leave it that way

Mergesort

Given an $array[a...b]$:

1. if $a = b$ leave it that way
2. $k = (a + b)/2$

Mergesort

Given an $array[a...b]$:

1. if $a = b$ leave it that way
2. $k = (a + b)/2$
3. sort recursively $array[a...k]$

Mergesort

Given an $array[a...b]$:

1. if $a = b$ leave it that way
2. $k = (a + b)/2$
3. sort recursively $array[a...k]$
4. sort recursively $array[(k + 1)...b]$

Mergesort

Given an $array[a...b]$:

1. if $a = b$ leave it that way
2. $k = (a + b)/2$
3. sort recursively $array[a...k]$
4. sort recursively $array[(k+1)...b]$
5. Merge the sorted subarrays into a sorted array $array[a...b]$

Mergesort

```
void sortVector(vector<int>& a,int l,int r){
    int m=(l+r)/2;
    if (l>=r) return;
    sortVector(a,l,m);
    sortVector(a,m+1,r);
    mergear(a,l,r,m);
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    vector<int> v={1,5,3,4,2,7,6,8};
    for(int i=0;i<8;i++) cout<<v[i]<<"\n "[i+1<8];
    sortVector(v,0,7);
    for(int i=0;i<8;i++) cout<<v[i]<<"\n "[i+1<8];
    exit(0);
}
```

Mergesort

```
void mergear(vector<int> &a, int l,int r,int m){
    vector<int> arreglado;
    int x=l;
    int y=m+1;
    while(x<=m||y<=r){
        if(x==m+1||(y<=r && a[y]<a[x])){
            arreglado.push_back(a[y]); y++;
        }
        else{
            arreglado.push_back(a[x]); x++;
        }
    }
    for(int i=l;i<=r;i++)a[i]=arreglado[i-l];
}
```

Decrease and Conquer

Decrease and conquer

- Some problems can be divided in only one subproblem
- for these reason they are called Decrease and conquer

Binary Exponentiation

- We want to calculate x^n
- We can define a build a function recursively
- This first function is slow (O(n)) in some problems
- We use decrease and conquer.

Binary Exponentiation

```
long long pot(int b,int e){
    if(e==0) return 1;
    long long x=pot(b,e/2)*pot(b,e/2);
    if(e%2) x=x*b;
    return x;
}
```

Binary search

Binary Search

- Given a **sorted** $array[a...b]$, we want to know if there is some particular element x.
    1. If the array is invalid($a > b$) return false

Binary Search

- Given a **sorted** *array*$[a...b]$, we want to know if there is some particular element x.
  1. If the array is invalid($a > b$) return false
  2. Let m=(a+b)/2

- Given a **sorted** $array[a...b]$, we want to know if there is some particular element x.
    1. If the array is invalid($a > b$) return false
    2. Let m=(a+b)/2
    3. If $array[m] = x$ return true
    4. If $array[m] < x$ the value x should be in $array[m + 1...b]$
    5. If $array[m] < x$ the value x should be in $array[a...m]$

Binary Search

```
bool binarysearch(vector<int> v,int lo,int hi, int x){
    if(lo>hi) return false;
    int m=(lo+hi)/2;
    if(v[m]==x) {
        /// process
        return true;
    }
    else if(v[m]<x) return binarysearch(v,m+1,hi,x);
    else if(v[m]>x) return binarysearch(v,lo,m-1,x);
}
```

Binary Search

```
bool binarysearch_v2(vector<int> v, int x){
    int lo=0,hi=v.size()-1;
    while(lo<=hi){
        int m=(lo+hi)/2;
        if(v[m]==x) return true;
        else if(x<v[m]){hi=m-1;}
        else if(x>v[m]) {lo=m+1;}
    }
    return false;
}
```

Binary Search over integers

- Given a predicate : $P : \mathbb{Z} \rightarrow \mathit{true}, \mathit{false}$ such that:
- $P(x) = \mathit{false} \forall x < n$, $P(x) = \mathit{true} \forall x \geq n$
- and **we want to find n**
- suppose that we have L and R with $P(L) = \mathit{false}$ and $P(R) = \mathit{true}$

Binary Search

```
while(L+1<R){
    int M=(L+R)/2;
    if(P(M)) R=M;
    else L=M;
}
```

Binary Search

```
while(L+1<R){
    int M=(L+R)/2;
    if(P(M)) R=M;
    else L=M;
}
```

```
while(L+1<R){
    int M=(L+R)/2;
    P(M) ? R=M: L=M;
}
```

Binary Search

```
while(L+1<R){
    int M=(L+R)/2;
    if(P(M)) R=M;
    else L=M;
}
```

```
while(L+1<R){
    int M=(L+R)/2;
    P(M) ? R=M: L=M;
}
```

- At the end of this code, we will get $n = L + 1$

Binary Search

```
while(L+1<R){
    int M=(L+R)/2;
    if(P(M)) R=M;
    else L=M;
}
```

```
while(L+1<R){
    int M=(L+R)/2;
    P(M) ? R=M: L=M;
}
```

- At the end of this code, we will get $n = L + 1$
- Complexity $O(log(R - L))$

Binary Search over reals

- Given a predicate : $P : \mathbb{R} \to true, false$ such that:
- $P(x) = false \forall x < T$, $P(x) = true \forall x \geq T$
- and **we want to find T**
- suppose that we have L and R with $P(L) = false$ and $P(R) = true$
- The bsi algorithm in this case would ocurre indefinitely
- But, for us it will be suffice to obtain a real T' such that $|abs(T' - T)| < \epsilon$

Binary Search over reals

```
double EPS = 1e-10,
L = -1000.0,
R = 1000.0;

while (R - L > EPS) {
    double mid = (L + R) / 2.0;
    if (p(mid)) R = mid;
    else L = mid;
}
```

Output decimal precision

- Be careful when you want to print a double:

Output decimal precision

- Be careful when you want to print a double:
- If you use *cout*, you can use
  $cout << fixed << setprecision(10) << L;$

Output decimal precision

- Be careful when you want to print a double:
- If you use *cout*, you can use
  $cout << fixed << setprecision(10) << L;$
- you can also use printf("%.10lf",L);

Gracias