Data Structures

Time Complexity and Formal Notations

Hikmat Farhat

May 31, 2018

Efficient Algorithms

- Given an algorithm to solve a problem we ask
- Is it efficient?
- We seek a sensible definition of efficiency
- How much work if the input doubles in size?
- For large input sizes can our algorithm solve the problem in a reasonable time?

Polynomial Time

- Efficient algorithm = polynomial in the size of the input

Definition

Polynomial Time: *for every input of size n*
$\exists a, b$ *such that number of computation steps* $< an^b$

- a and b are constants that do not depend on n
- True, some algorithms are polynomials with a and/or b very large
- but for the majority of algorithms, a and d are relatively small

Why Polynomial Time ?

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Worst Case Analysis

- Usually the running time is the running time of the worst case
- One could analysis the average case but it much more difficult and depends on the chosen distribution.
- Therefore an algorithm is efficient if it has a worst case polynomial time
- There are exceptions the most important being the simplex algorithm that works very well in practice

Informal Example: Union Find

- We will introduce the cost of algorithms informally by an example: union find.
- We have a set of n points and a set of m connections between these points.
- For any two points p and q we would like to answer the questions: is there a path from p to q?
- Three different algorithms, with different costs, will be presented to solve the above problem.

Union Find: attempt number 1

- The basic idea is to associate an identifier with every point, so we maintain an array $id[n]$.
- The identifier of a given point is the group the point belongs to. Initially there are n group with one point in each, namely $id[i] = i$
- When two points, p and q, are found to be connected their respective groups are merged (union).

The find function

- the function Find(p) returns the group id that p belongs to

Instuctions	*cost*	*times*
Find (p)	c_0	1
Return $id[p]$	c	1

- therefore the cost of function find(p) is constant (i.e. independent of the number of points)

Instructions	*cost*	*times*
Union (p,q)	c_0	1
$idp \leftarrow$ Find(p)	c_1	1
$idq \leftarrow$ Find(q)	c_1	1
if $idp = idq$ **then**	c_2	1
Return	c_3	1
for $i = 0$ **to** $n-1$ **do**	c_4	$n+1$
if $id[i] = idp$ **then**	c_5	n
$id[i] \leftarrow idq$	c_6	t_p
$count \leftarrow count - 1$	c_7	1
Return	c_3	1

- Whe $p = q$ the cost of Union is $c_0 + 2c_1 + c_2 + c_3 = A$ and when $p \neq q$ the total cost of Union is

$$c_0 + 2c_1 + c_2 + c_4(n+1) + c_5 n + c_6 t_p + c_7 + c_3$$

- Which can be written as $B + Cn + c_6 t_p$ where B and C are constants and t_p is evaluated next

Computational Cost

- How much does it "cost" to run Find and Union when we have n points?
- For Find we already calculated it, it is a constant, $d_0 + d_1$, independent of the number of points.
- For Union, we still need to calculated t_p which is the number of times line 8 is executed.
- Line 8 is executed **at least** once since we know that at least $id[p] = idp$.
- Also, line 8 is executed **at most** $n - 1$ times because at least $id[q] \neq idp$.
- Therefore $1 \leq t_p \leq n - 1$. Inserting the value of t_p in the previous calculation for Union we get:
  $B + Cn + c_6 \leq Cost \leq (B - c_6) + (C + c_6)n$
- Rearranging terms we get

$$\alpha + \beta n \leq Cost \leq \gamma + \delta n$$

Quick Union

- A different approach is to organize all related points in a tree structure.
- Two points belong to the same group iff they belong to the same tree.
- A tree is uniquely identified by its root.
- The array $id[]$ has a different meaning: $id[i] = k$ means that site k is the parent of site i.
- Only the root of a tree has the property $id[i] = i$;
- In this case $find(p)$ returns the root of the tree that p belongs to.

Quick Union Pseudo Code

```
Find(p)
```

while $id[p] \neq p$ **do**
   $p = id[p]$
end
return p

```
Union(p,q)
```

$proot \leftarrow \texttt{Find}(p)$
$qroot \leftarrow \texttt{Find}(q)$
if $proot = qroot$ **then**
   return
end
$id[proot] \leftarrow qroot$
$count \leftarrow count - 1$

Quick Union Cost

- The cost of $Find(p)$ is $2d + 1$ where d is the depth of node p.
- This means that the cost of $Union(p, q)$ is between $(2d_p + 1) + (2d_q + 1) + 1$ and $(2d_p + 1) + (2d_q + 1) + 3$.
- The problem is that in some cases the tree degenerates into a linear list.
- In that case the height=size and instead of getting $\log n$ behavior we get n.
- To avoid such a situation we try to keep the trees **balanced**.
- We do this by always attaching the small tree to the large one.
- to this end we introduce a variable $tsize$ initialized to 1.

Union Find: take 3

`Union(`p,q`)`

$proot \leftarrow$ `Find`(p)
$qroot \leftarrow$ `Find`(q)
if $proot = qroot$ **then**
    return
end
if $tsize[proot] > tsize[qroot]$ **then**
    $id[qroot] \leftarrow proot$
    $tsize[proot] \leftarrow tsize[proot] + tsize[qroot]$
else
    $id[proot] \leftarrow qroot$
    $tsize[qroot] \leftarrow tsize[proot] + tsize[qroot]$
end
$count \leftarrow count - 1$

Weighted Quick Union

- We have shown that in the quick union version UnionFind, "find(p)" costs $2d + 1$ where d is the depth of node p.
- we will now show that during the computation of the weighted quick union for N sites, the depth of ANY node is at most $\log N$.
- It is sufficient to show that the height of ANY tree of size k is at most $\log k$ (this is not the case in the original quick union where the height can be up to $k - 1$)

Proof

- By induction on the size of the tree.
- Base case: $k = 1$ then there is only one node and the height is $\log k = 0$.
- Assume that for any tree, T of size $i < n$, the height of T, h_i is at most $\log i$ and consider two trees of size $i \leq j$.
- So we have $h_i \leq \log i$ and $h_j \leq \log j$.

- The size of the combined tree is $i + j = k$.
- Using the weighted quick union method we know that the height of the combined tree is at most $max(1 + \log i, \log j)$ (why?)
- in the first case $1 + \log i = \log 2i \leq \log(i + j) = \log k$
- in the second case $\log j \leq \log(i + j) = \log k$.

Asymptotic Growth of Functions

Definition

Big Oh: The set $O(g(n))$ is defined as all functions $f(n)$ with the property $\exists c, n_0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$

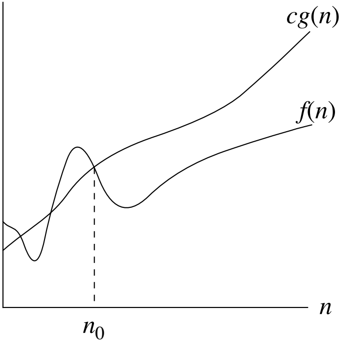


Figure : Graphical definition of O taken from the CLRS book

Example

- $f(n) = 2n^2 + n = O(n^2)$ because let $c = 3$ and $n_0 = 1$

$$
\begin{aligned}
\forall n \geq n_0 = 1 & \\
n &\leq n^2 \\
2n^2 + n &\leq 3n^2 \\
f(n) &\leq cn^2 \\
f(n) &= O(n^2)
\end{aligned}
$$

- On the other hand $f(n) = 2n^2 + n \neq O(n)$

Definition

Big Omega: *The set $\Omega(g(n))$ is defined as all functions $f(n)$ with the property*
$\exists c, n_0$ *such that* $f(n) \geq cg(n)$ *for all* $n \geq n_0$

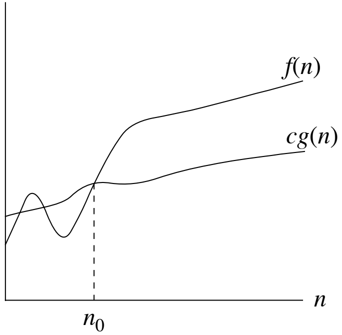


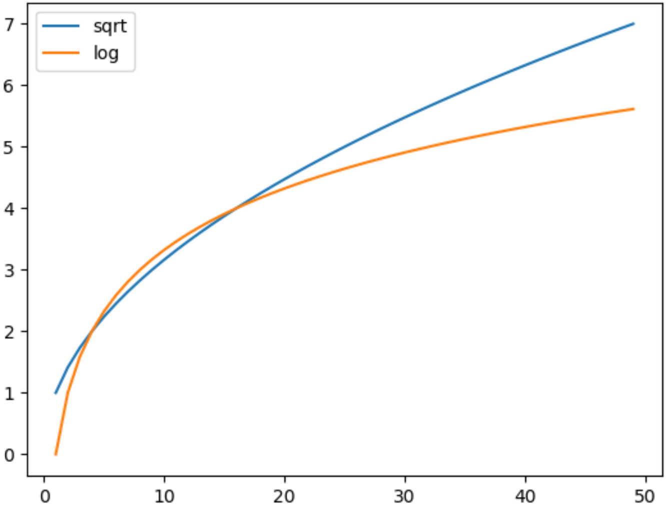
Figure : Graphical definition of Ω taken from the CLRS book

Example

- Consider $f(n) = \sqrt{n}$ and $g(n) = \log n$.
- $f(n) = \Omega(g(n))$ because for $c = 1, n_0 = 16$ we have

$$\sqrt{16} = 4 = \log 2^4$$

- Note that between n=4 and n=16 the value of $\log_2 n \geq \sqrt{n}$


sqrt
log
7
6
5
4
3
2
1
0
0
10
20
30
40
50

- Abuse of notation: if $h(n) \in O(g(n))$ we write $h(n) = O(g(n))$
- similarly if $h(n) \in \Omega(g(n))$ we write $h(n) = \Omega(g(n))$
- If $h(n) = O(g(n))$ we say $g(n)$ is an <u>upper bound</u> for $f(n)$.
- If $h(n) = \Omega(g(n))$ we say $g(n)$ is a <u>lower bound</u> for $f(n)$.

Definition

Big Θ: *The set* $\Theta(g(n))$ *is defined as all functions* $f(n)$ *with the property* $\exists c_1, c_2, n_0$ *such that* $c_1 g(n) \le f(n) \le c_2 g(n)$ *for all* $n \ge n_0$

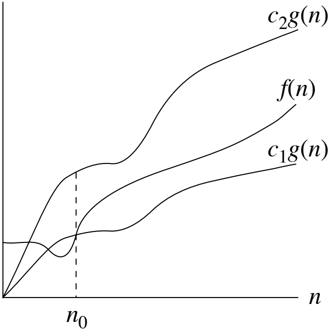


Figure : Graphical definition of Θ taken from the CLRS book

Example

- Consider $c_1 = 1$, $c_2 = 3$ and $n_0 = 1$ it is obvious that

$$c_1 n^2 \leq 2n^2 + n \leq c_2 n^2 \ \forall n \geq n_0$$

- We can show that $f(n) = \Theta(g(n))$ iff
  $f(n) = O(g(n))$ and
  $f(n) = \Omega(g(n))$
- If $f(n) = \Theta(g(n))$ we say $g(n)$ is a <u>tight bound</u> for $f(n)$.

Definition

Little Oh: The set $o(g(n))$ is defined as all functions $f(n)$ with the property for all c, $\exists n_0$ such that $f(n) < cg(n)$ for all $n \geq n_0$

We can show that the above definition is equivalent to

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Definition

Little omega: The set $\omega(g(n))$ is defined as all functions $f(n)$ with the property
for all c, $\exists n_0$ such that
$cg(n) < f(n)$ for all $n \geq n_0$

We can show that the above definition is equivalent to

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

Examples

- $f(n) = 2n^2 + n$
- $f(n) = \omega(n)$ because

$$\lim_{n\to\infty} \frac{2n^2 + n}{n} = \infty$$

- $f(n) = o(n^3)$ because

$$\lim_{n\to\infty} \frac{2n^2 + n}{n^3} = 0$$

Using Limits

- Sometimes it is easier to determine the relative growth rate of two functions $f(n)$ and $g(n)$ by using limits
  - if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ then $f(n) = o(g(n))$.
  - if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ then $f(n) = \omega(g(n))$.
  - if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, for some constant c, then $f(n) = \Theta(g(n))$.
- Can we always do that?No
- In many situations the complexity cannot be written in an analytic form.

Exponential vs Polynomial vs Logarithmic

it is easy to show that for all $a > 0$,$b > 1$

$$\lim_{n\to\infty} \frac{n^a}{b^n} = 0$$

And polynomials grow faster(use $m = \log n$ and the previous result) than logarithms ($\log^a x = (\log x)^a$)

$$\lim_{n\to\infty} \frac{\log^a n}{n^b} = 0$$

Arithmetic Properties

- transitivity: if $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$.
  - eg: $\log n = O(n)$ and $n = O(2^n)$ then $\log n = O(2^n)$.
- constant factor: if $f(n) = O(kg(n))$ for some $k > 0$ then $f(n) = O(g(n))$.
  - eg: $n^2 = O(3n^2)$ thus $n^2 = O(n^2)$.
- sum: if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$
  - $3n^2 = O(n^2)$, $6 \log n = O(\log n)$ then $3n^2 + 6 \log n = O(n^2)$
- product: $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n) * f_2(n) = O(g_1(n) * g_2(n))$
  - eg: $3n^2 = O(n^2)$, $6 \log n = O(\log n)$ then $3n^2 * 6 \log n = O(n^2 \log n)$

Code Fragments

$sum = 0$
for $i = 1 \dots n$ **do**
  $sum \leftarrow sum + 1$
end

- the operation $sum = 0$ is independent of the input thus it costs a constant time c_1.
- the operation $sum \leftarrow sum + 1$ is independent of the input thus it cost some constant time c_2.
- Regardless of the input the loop runs n times therefore the total cost is $c_1 + c_2 n = \Theta(n)$.

The algorithm below for finding the maximum of n numbers is $\Theta(n)$.
Input: $a_1, \ldots, a_n$
Output: $max(a_1, \ldots, a_n)$

Initially $max = a_1$
for $i = 2 \ldots n$ **do**
    if $a_i > max$ **then**
        $max = a_i$
    end
end

- Try 89,47,80,50,67,102 and 19,15,13,10,8,3
- Do they have the "same" running time?

Sequential Search

Given an array a check if element x is in the array.

for $i = 1 \ldots n$ **do**
  if $a[i] = x$ **then**
    return True
  end
end
return False

- What is the running time of the above algorithm?
- Consider the two extreme cases: x is the first or the last element of the array.

- If x is the first element than we perform a single operation. This is the best-case.
- If x is the last element than we perform n operation.This is the worst-case.
- Now if we run the algorithm on many different (random) input and average out the results we get the average-case.
- Which one do we use?
  - Depends on the application and the feasibility.
  - Real-time and critical applications usually require worst-case
  - In most other situations we prefer average-case, but difficult to calculate and depends on the random distribution!
- In light of the above, what is the best-case,average-case and worst-case for the compute max algorithm we had before?

Nested Loops

- What is the complexity of nested loops?
- The cost of a stmt in a nested loop is the cost of the statement multiplied by the product of the size of the loops

for $i = 1 \ldots n$ **do**
  for $j = 1 \ldots m$ **do**
    $k \leftarrow k + 1$
  end
end

- The cost is $O(n * m)$

Factorial

- Consider the recursive implementation of the factorial function.

```
factorial(n)

if n=1 then
|  return 1
else
|  return n*factorial(n-1)
end
```

- The cost of size n? $T(n) = T(n-1) + C$
- Thus $T(n) = \Theta(n)$.

Complexity of Factorial

$$
\begin{aligned}
T(n) &= T(n-1) + C \\
&= T(n-2) + 2C \\
&= \dots \\
&= T(n-i) + i * C \\
&= \dots \\
&= T(1) + (n-1) * C \\
&= n * C + T(1) - C \\
&= \Theta(n)
\end{aligned}
$$

Fibonacci

- Computing the n^{th} Fibonacci number can be done recursively

```
fib(n)
```

if $n = 0$ **then**
  return *0*
end
if $n = 1$ **then**
  return *1*
end
return `fib(n-1)+fib(n-2)`

- If $T(n)$ is the cost of computing $Fib(n)$ then

$$T(n) = T(n-1) + T(n-2) + 3$$

- We will show, by **induction** on n, that $T(n) \geq fib(n)$ i.e. the **cost** of computing Fibonacci number n is greater than the **number** itself.

- We are assuming that all operations cost the same so the 3 comes from executing the two if stmts and the sum.
- First the base cases. If $n = 0$ then the algorithm costs 1 (if stmt), if $n = 1$ it costs 2 (2 if stmts) thus $T(0) = 1,\ T(1) = 2$.
- In the other cases we have $T(n) = T(n-1) + T(n-2) + 3$. This means $T(2) = 6 \geq fib(2) = 1$.
- Assume that $T(n) \geq fib(n)$ then

$$
\begin{aligned}
T(n+1) &= T(n) + T(n-1) + 3 \\
&\geq fib(n) + fib(n-1) && \text{hyp.} \\
&\geq fib(n+1)
\end{aligned}
$$

- One can show that (for $n \geq 5$) $fib(n) \geq (3/2)^n$ thus $T(n) \geq (3/2)^n$ which is exponential!

Fibonacci: take two

- can we compute Fibonacci numbers more efficiently?
- It turns out yes. By just "remembering" the values we already computed.
- A simple iterative algorithm

FiboIter(n)
$f[0] \leftarrow 0$
$f[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
    $f[i] \leftarrow f[i-1] + f[i-2]$
end

Comparison

- We had two different algorithms to compute Fibonacci number n
- One was $\Omega((3/2)^n)$ while the other was $O(n)$.
- In the first one we did not need to "save" anything.
- In the second algorithm we used an array of size n: **space complexity**: $O(n)$.
- This is a **trade off** between time and space.
- Obviously in this case the trade off is worth it.

Exponentiation

- Exponentiation is another example where the simplest algorithm is much less than optimal.
- The simplest way to compute x^n is $\underbrace{x \dots x}_{n \text{ times}}$
- therefore the complexity of the above algorithm is $\Theta(n)$.
- We can do (much) better by observing that
- if n is even then $x^n = (x^{n/2})^2$
- if n is odd then $x^n = (x^{n/2})^2 \cdot x$
- Note the integer division, $n/2 \equiv \lfloor n/2 \rfloor$, e.g. $7/2{=}3$

Implementation

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

Complexity of Exponentiation

- The analysis is simplified by assuming $n = 2^k$ (other cases are similar, albeit more complicated)
- Assume: $n/2$, $half * half$ and the if stmt each costs 1.
- for a total of 4 (including the test for x==0) when n is even and 5 when it is odd.
- Let $T(n)$ be the computational cost for x^n then

$$
\begin{aligned}
T(n) &= T(n/2) + 4 \\
&= T(n/4) + 8 \\
&= T(n/2^i) + 4i \\
&= \dots \\
&= T(1) + 4k \\
&= \Theta(k) = \Theta(\log n)
\end{aligned}
$$

Exponentiation

- In the general case we perform one extra computation every time the exponent is odd.
- Let $\beta(n)$ be the number of times such computation is performed.
- It is easy to check that $\beta(n) =$ is one less than the number of 1's in the binary representation of n
- For example if $n = 21$ then $21 \to 10 \to 5 \to 2 \to 1$ which means the intermediate value is odd twice.
- Compare with the binary representation $21 = 1011$.
- Clearly $\beta(n)$ is at most equal to the number of bits in the binary representation of n which is $\lfloor \log n \rfloor$
- So even in the general case the complexity is $\Theta(\log n)$.

General case

- In general the problem has the following recursion relation

$$T(n) = T(\lfloor n/2 \rfloor) + 4 + (n \mod 2)$$

- We will show that the general form

$$T(n) = T(\lfloor n/2 \rfloor) + M + (n \mod 2) \tag{1}$$

- Has solution

$$T(n) = M\lfloor \log n \rfloor + \beta(n) \tag{2}$$

- Where $\beta(n)$ is the number of 1's in the binary representation of n.
- Using the fact that $\lfloor \log \lfloor n/2 \rfloor \rfloor = \lfloor \log n \rfloor - 1$ it is easy to check that (2) satisfies (1).

Fibonacci: Take Three

- The previous method for exponentiation can be used to compute Fibonacci (n) in $O(\log n)$.
- The key is that

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

- The above is shown by induction on n.
- We know how to compute the power in $\log n$.

Maximum Subarray Sum

- Given an array A of n elements we ask for the maximum value of

$$\sum_{k=i}^{j} A_k$$

- For example if A is -2,11,-4,13,-5,-2 then the answer is $20 = \sum_{k=2}^{4}$

Brute Force

- Compute the sum of all subarrays of an array A of size n and return the largest.
- A subarray starts at index i and ends at index j where $0 \leq i < n$ and $0 \leq j < n$.
- Therefore for **each possible** i and j compute the sum of $A[i] \ldots A[j]$.

```
int maxSubarray(int *A,int n){
  int sum=0, max=A[0];

  for(int i=0;i<n;i++){
      for(j=i;j<n;j++){
         sum=0;
         for(int k=i;k<=j;k++)
              sum+=A[k];
         if(max<sum)max=sum;
      }
  }
return max;
}
```

Complexity

- To determine the complexity of the brute force approach we can see that there are 3 nested loop therefore the complexity of the problem depends on how many times line 14 is executed
- The number of executions is

$$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}\sum_{k=i}^{j} 1 = \sum_{i=0}^{n-1}\sum_{j=i}^{n-1} j - i + 1$$

- To evaluate the first sum let $m = j - i + 1$ then

$$\sum_{j=i}^{n-1} j - i + 1 = \sum_{m=1}^{n-i} m = (n-i)(n-i+1)/2$$

- Finally, we get

$$\begin{aligned}\sum_{i=0}^{n-1}(n-i)(n-i+1)/2 &= \frac{n^3+3n^2+2n}{6} \\ &= \Theta(n^3)\end{aligned}$$

Divide and Conquer

- general technique that divides a problem in 2 or more parts (divide) and patch the subproblems together (conquer).
- In this context if we divide an array in two subarrays. We have 3 possibilities:
    1. max is entirely in the first half
    2. max is entirely in the second half
    3. max spans both halves.
- Therefore the solution is max(left,right,both)

Both halves

- If the sum spans both halves it means it includes the last element of the first half and the first element of the second half
- This means that the we are looking for the sum of
    1. Max subsequence in first half that includes the last element
    2. Max subsequence in the second half that includes the first element

$$
\begin{aligned}
S_3 &= \max_{\substack{0 \le i < n/2 \\ n/2 \le j < n}} \sum_{k=i}^{j} A[k] \\
&= \max_{\substack{0 \le i < n/2 \\ n/2 \le j < n}} \left[\sum_{k=i}^{n/2-1} A[k] + \sum_{k=n/2}^{j} A[k] \right] \\
&= \max_{0 \le i < n/2} \sum_{k=i}^{n/2-1} A[k] + \max_{n/2 \le j < n} \sum_{k=n/2}^{j} A[k]
\end{aligned}
$$

Computing max that spans both halves

`computeBoth` (A,left,right)

$sum_1 \leftarrow sum_2 \leftarrow 0$
for $i = center$ **to** $left$ **do**
    $sum_1 \leftarrow sum_1 + A[i]$
    if $sum_1 > max_1$ **then**
        $max_1 \leftarrow sum_1$
    end
end
for $j = center + 1$ **to** $right$ **do**
    $sum_2 \leftarrow sum_2 + A[j]$
    if $sum_2 > max_2$ **then**
        $max_2 \leftarrow sum_2$
    end
end
return $max_1 + max_2$

Recursive Algorithm

`maxSubarray`$(A, left, right)$

$center \leftarrow (left + right)/2$
$S_1 \leftarrow \texttt{maxSubarray}(A, left, center)$
$S_2 \leftarrow \texttt{maxSubarray}(A, center + 1, right)$
$S_3 \leftarrow computeBoth(A, left, right)$
return $\max(S_1, S_2, S_3)$

Complexity

- Given an array of size n the cost of the call to *maxSubarray* is divided into two computations
  1. The work of *computeBoth* which is $\Theta(n)$.
  2. **Two** recursive calls on the problem with **half the size**
  3. Therefore the total cost can be written as

$$T(n) = 2T(n/2) + \Theta(n)$$

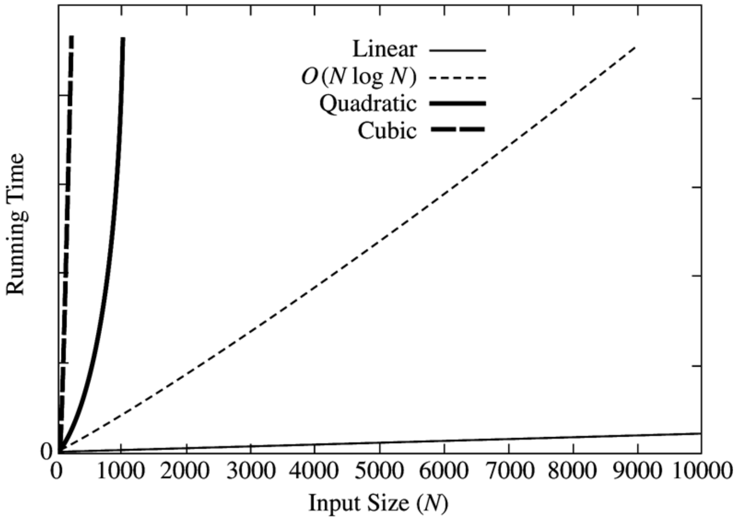
- to solve the above recurrence, we assume for simplicity that $n = 2^k$

- Thus

$$
\begin{aligned}
T(2^k) &= 2T(2^{k-1}) + C \cdot 2^k \\
&= 2(2T(2^{k-2}) + 2^{k-1}) + C \cdot 2^k \\
&= 2^2 T(2^{k-2}) + 2 \times C \cdot 2^k \\
&= \ldots\ldots \\
&= 2^i T(2^{k-i}) + i \cdot C \cdot 2^k \\
&= 2^k T(1) + k \cdot C \cdot 2^k \\
&= \Theta(n \log n)
\end{aligned}
$$

Running time comparison

- There *is* an $\Theta(n)$ algorithm for max subarray. Can you find it?



Master Theorem (special case)

- A generalization of the previous cases is done using a **simplified** version of the Master theorem

$$T(n) = aT(n/b) + \Theta(n^d)$$

$$
\begin{aligned}
T(n) &= aT(n/b) + cn^d \\
&= a\left[aT(n/b^2) + c(n/b)^d\right] + cn^d \\
&= a^2T(n/b^2) + cn^d(a/b^d) + cn^d \\
&= a^2\left[aT(n/b^3) + c(n/b^2)^d\right] + cn^d(a/b^d) + cn^d \\
&= a^3T(n/b^3) + cn^d(a/b^d)^2 + cn^d(a/b^d) + cn^d \\
&= a^iT(n/b^i) + cn^d\sum_{l=0}^{i-1}(a/b^d)^l
\end{aligned}
$$

The above reaches $T(1)$ when $b^k = n$ for some k. We get

$$T(n) = a^k T(1) + cn^d \sum_{l=0}^{k-1} (a/b^d)^l$$

There are three cases

1. $a = b^d$
2. $a < b^d$
3. $a > b^d$

case 1: $a = b^d$

If $a = b^d$ (i.e $\frac{a}{b^d} = 1$) then we get

$$T(n) = a^k T(1) + cn^d \cdot k$$

Since $k = \log_b n$ then

$$\begin{aligned} T(n) &= a^{\log_b n} T(1) + cn^d \log_b n \\ &= n^{\log_b a} T(1) + cn^d \log_b n \\ &= n^d T(1) + cn^d \log_b n \\ &= \Theta(n^d \log n) \end{aligned}$$

case 2: $a < b^d$

$$T(n) = a^k T(1) + cn^d \sum_{l=0}^{k-1} (a/b^d)^l$$
$$= a^k T(1) + cn^d \frac{(a/b^d)^k - 1}{(a/b^d) - 1}$$

for large n, i.e. $n \to \infty$ then $k = \log_b n \to \infty$ and since $a < b^d$ then $a/b^d \to 0$ Therefore

$$T(n) = n^{\log_b a} T(1) + cn^d$$

but $a < b^d \Rightarrow \log_b a < d$ and finally

$$T(n) = \Theta(n^d)$$

case 3: $a > b^d$

In this case we can write

$$
\begin{aligned}
T(n) &= a^k \frac{a}{b} T(n) \\
&= n^{\log_b a} T(1) + gn^d (a/b^d)^k \\
&= n^{\log_b a} T(1) + gn^d (a/b^d)^{log_b n} \\
&= n^{\log_b a} T(1) + gn^d n^{log_b (a/b^d)} \\
&= n^{\log_b a} T(1) + gn^d n^{(-d + log_b a)} \\
&= \Theta(n^{\log_b a})
\end{aligned}
$$

$$
= a^k T(1) + cn^d \frac{(a/b^d)^k - 1}{(a/b^d) - 1}
$$