CS2521: Divide and Conquer

N. Oren
n.oren@abdn.ac.uk

University of Aberdeen

Binary search — redux

```
function binarySearch(A,k,l,h)
    if l>h then return -1
    m=(l+h)/2
    if A[m]==k then return middle
    if A[m]>k then return binarySearch(A,k,l,m-1)
    else return binarySearch(A,k,m+1,h)
end function
```

- Binary search finds an item in a list in $\Theta(\log n)$ time.
- Very efficient: $2^{20} \approx 10^6$ so we can find a key in a million element array within 20 time steps.
- Can we adapt binary search to do other things?

Binary search — redux

```
function binarySearch(A,k,l,h)
    if l>h then return -1
    m=(l+h)/2
    if A[m]==k then return middle
    if A[m]>k then return binarySearch(A,k,l,m-1)
    else return binarySearch(A,k,m+1,h)
end function
```

- What if we want to know the number of occurrences of a key in an array?
- Binary search for it, and search left and right from the key until a different element is found.
- Complexity?

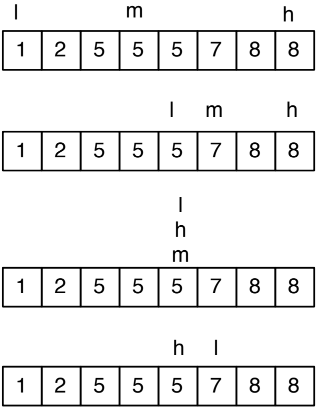
Binary search — redux

```
function binarySearch(A,k,l,h)
    if l>h then return -1
    m=(l+h)/2
    if A[m]==k then return middle
    if A[m]>k then return binarySearch(A,k,l,m-1)
    else return binarySearch(A,k,m+1,h)
end function
```

- What if we want to know the number of occurrences of a key in an array?
- Binary search for it, and search left and right from the key until a different element is found.
- Complexity? $O(s + \log n)$. If $|s| = n$ then linear complexity.
- We can do better.

Binary search — redux

```
function findEnd(A,k,l,h)
    if l>h then return l
    m=(l+h)/2
    if A[m]>k then return findEnd(A,k,l,m-1)
    else return findEnd(A,k,m+1,h)
end function
```



Binary search — redux

```
function findEnd(A,k,l,h)
    if l>h then return l
    m=(l+h)/2
    if A[m]>k then return findEnd(A,k,l,m-1)
    else return findEnd(A,k,m+1,h)
end function
```

- this function will always find the rightmost element of the key.
- Reversing the function (**homework**) allows us to find the leftmost element.
- Subtracting the indexes will tell us how many elements we have.
- Complexity: $2\log n = O(\log n)$

Applications of Binary Search

- Consider a huge array consisting of a bunch of 0's followed by 1's till the end.
- Can we find the transition point efficiently?
- One-sided binary search, starting at the current index and searching $A[i+1], A[i+2], A[i+4] \ldots$ allows us to identify a window in which we can find some transition element. We can then do a binary search within this window to find the transition point regardless of array size.
- The complexity of this function is defined in terms of the position of the transition point rather than the size of the array.
- At most $2\lceil \log p \rceil$ comparisons.

Something slightly different

- How can we find the square root of some number n?
- $\sqrt{n} = r \rightarrow n = r^2 \rightarrow r^2 - n = 0$
- What can we do without a calculator?
- We know the square root of any number must be greater than 0 and at most n (we can obviously tighten these bounds).
- Let $l = 0, h = n$
- Try $m = (l + h)/2$ if $m * m > n$, then m is too large, try again with $h = m$. Otherwise, our guess is too small, try again after setting $l = m$.
- Each time, we're cutting our search interval in half.
- This bisection method can be applied to any function as long as $f(l) > 0$ and $f(r) < 0$. to find x such that $f(x) = 0$

Divide and Conquer

- These examples, as well as Mergesort and quick sort are examples of divide and conquer algorithms.
- Such an algorithm splits a problem into a smaller set of problems, solves them, and then combines the solutions to find a solution to the full problem.
- If combining takes less time than solving the entire problem, then we've made an efficiency gain.

Divide and Conquer

```
function dac(input)
    if |input|=1 then return solve(input)
    sp_{1...k}=split(input)
    for all k do
        ssn_k = dac(sp_k)
    end for
    return combine(ssn_1, ..., ssn_k)
end function
```

- How do we compute the complexity of a D&C algorithm?
- Let's say that the combine operation takes time $f(n)$ over input size n.
- And that for an input of size 1, solving is $\Theta(1)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } |n| = 1 \\ kT(n/k) + f(n) & \textit{otherwise} \end{cases}$$

- We typically ignore the $\Theta(1)$, writing $T(n) = kT(n/k) + f(n)$.

Divide and Conquer

```
function dac(input)
    if |input|=1 then return solve(input)
    sp_{1...k}=split(input)
    for all k do
        ssn_k = dac(sp_k)
    end for
    return combine(ssn_1, ..., ssn_k)
end function
```

- More generally, a D&C problem typically breaks some problem into a smaller pieces, each of which is of size n/b (note, b does not have to equal a).
- Combining the subproblems takes time $f(n)$
- Then $T(n) = aT(n/b) + f(n)$
- $T(n)$ is defined in terms of its values on smaller instances of itself.
- This is called a recurrence.

Rabbit Breeding

- Assume you start with one pair of newborn rabbits (a male and a female), and in every month following this, each pair of rabbits that are more than a month old give birth to a new male and female rabbit. How many pairs of rabbits do you have at the end of n months?

Rabbit Breeding

- Assume you start with one pair of newborn rabbits (a male and a female), and in every month following this, each pair of rabbits that are more than a month old give birth to a new male and female rabbit. How many pairs of rabbits do you have at the end of n months?
- $T(0) = 1$
- $T(1) = 1$

Rabbit Breeding

- Assume you start with one pair of newborn rabbits (a male and a female), and in every month following this, each pair of rabbits that are more than a month old give birth to a new male and female rabbit. How many pairs of rabbits do you have at the end of n months?
- $T(0) = 1$
- $T(1) = 1$
- $T(n) = T(n-1) + T(n-2)$
- 1,1,2,3,5,8, ...
- This is known as the Fibonacci series

Examples

- Mergesort: $T(n) = 2T(n/2) + O(n)$
  - we broke the algorithm into 2 equal sized halves and spent $O(n)$ time combining them.
  - The recurrence evaluates to $T(n) = O(n \log n)$
- Binary search: $T(n) = T(n/2) + O(1)$
  - At each step we spend constant time to reduce the problem to half its size (and only evaluate one of the two halves).
  - The recurrence evaluates to $T(n) = O(\log n)$
- So why do we care?

Examples

- Mergesort: $T(n) = 2T(n/2) + O(n)$
  - we broke the algorithm into 2 equal sized halves and spent $O(n)$ time combining them.
  - The recurrence evaluates to $T(n) = O(n \log n)$
- Binary search: $T(n) = T(n/2) + O(1)$
  - At each step we spend constant time to reduce the problem to half its size (and only evaluate one of the two halves).
  - The recurrence evaluates to $T(n) = O(\log n)$
- So why do we care?
- Matrix Multiplication
  - naïve algorithm is $O(n^3)$.
  - Strassen's algorithm breaks a matrix into 7 $n/2 \times n/2$ matrices and combines them in time $O(n^2)$.
  - The recurrence evaluates to $T(n) \approx O(n^{2.81})$
  - This dominates the $O(n^3)$ solution!
- Without being able to solve recurrences, we wouldn't know this.

Solving Recurrences

- There are 3 basic approaches to solving recurrences
  - Via the master theorem
  - The recursion-tree method
  - The substitution method

Solving Recurrences

- There are 3 basic approaches to solving recurrences
  - Via the master theorem
  - The recursion-tree method
  - The substitution method (a.k.a guessing)

The Substitution Method

- The substitution method consists of 2 steps.
  1. Guess the form of the solution
  2. Prove the solution correct
- The name comes from our substituting the guessed solution for the function during the proof step.

An Example

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

- We observe this is very similar to the mergesort recurrence $(2T(n/2) + O(n))$, so we guess

$$T(n) = O(n \log n)$$

- Going back to basic definition of O notation, we must show that $T(n) \leq cn \log n$ for some constant $c > 0$

An Example

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \qquad Guess: T(n) = O(n \log n)$$

- Check base case ($T(1) = 1$):

$$T(1) \leq c1 \log 1 = 0$$

- uh oh...
- But we assume that the relation holds for all $n > n_0$.
- And we can see that if $n > 3$, the relation doesn't actually depend on $T(1)$.
- So if we can verify for $T(2)$ and $T(3)$ (as these are the only ones that reduce to $T(1)$), we're ok.

$$\begin{aligned} T(2) &= (2)(1) + 2 = 4 \\ &\leq c2 \log 2 \\ &= 2c \end{aligned} \qquad \begin{aligned} T(3) &= 2(1) + 3 = 5 \\ &\leq c3 \log 3 \\ &\approx 4.75c \end{aligned}$$

- Base case ok (for $c > 2$)!

An Example

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \qquad Guess : T(n) = O(n \log n)$$

- Inductive assumption: assume that this holds for all values up to $m = \lfloor n/2 \rfloor$
- We need to show that it holds for the next step of the recurrence, i.e. for n.
- According to our inductive assumption

$$T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$$

- So let's substitute it in...

$$T(n) \leq 2(c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n$$

An Example

$$T(n) \leq 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n$$

- Now we simplify.

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\ &\leq cn \log(n/2) + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &\leq cn \log n \end{aligned}$$

- The last step holds as long as $c \geq 1$

What have we just done?

- We have proved by induction that an algorithm whose running time can be described by the recurrence $T(n) = 2T(\lfloor n/2 \rfloor) + n$ is $O(n \log n)$
- How?
  1. Proven the base case
  2. Assumed true for some value (usually the one found in the recurrence).
  3. Shown true for a greater value (namely the next one in the recurrence).

What if we had made a mistake?

- What if we had tried $T(n) = O(n)$?
- Base case still OK.
- Substitute in:

$$\begin{aligned} T(n) &\leq 2c\lfloor n/2 \rfloor + n \\ &\leq cn + n \\ &= (c+1)n \end{aligned}$$

- Now this does not imply that $T(n) \leq cn$, but rather something greater. So this is the wrong guess.

Another Example

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

- To make a guess, let's expand a bit.

$$T(3) = 2T(2) + d \qquad T(2) = 2T(1) + d$$

- So $T(1) = 1$, $T(2) = 2 + d$, $T(3) = 4 + 3d$, $T(4) = 8 + 7d$. This looks suspiciously like 2^n. Let's try that, noting that we have a constant in there...
- Our guess is $T(n) = O(2^n)$.
- Base case: $T(1) = 1 \leq 2^1$. So that's fine
- Inductive case. Assume true for $n-1$, let's check for n

$$\begin{aligned} T(n) &\leq 2(c2^{n-1}) + d \\ &= c2^n + d \end{aligned}$$

- uh oh... wrong guess.

Fixing the guess

- Let's instead assume that that $T(n) \leq 2^n - b$ where b is a constant.
- Base case: $T(1) = 1 \leq c2^1 - b$, which is ok for $c \geq (b+1)/2$.
- Inductive case:

$$\begin{aligned} T(n) &\leq 2(c2^{n-1} - b) + d \\ &= c2^n - 2b + d \\ &\leq c2^n - b \text{ if } b \geq d \end{aligned}$$

- This holds for all c, as long as c is consistent with our base case.
- So we have shown that $T(n) \leq c2^n - b$ for $b \geq d, c \geq (b+1)/2$
- And since $c2^n - b = O(2^n)$ so is $T(n)$
- Take home message: we are allowed to substitute in lower order terms, these can make solving the recurrence easier!

Guessing Well

- So far, we've come up with guesses based on recurrences we've encountered before.
- A good way to generate a good guess is via a recursion tree.
- In such a tree, a node represents the cost of solving a single subproblem.
- We sum the costs within each level of the tree to obtain a set of per-level costs.
- Summing all of these gives us the total cost of the recursion.

Example

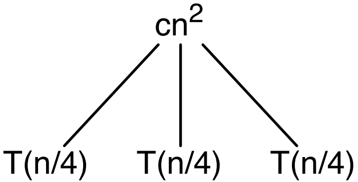
$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$

- Recursion trees provide us with "good guesses", we can be a little sloppy.
- We drop the $\lfloor$ and $\rfloor$, creating a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$.
- The c appears due to the definition of Θ, and is a constant greater than 0.
- For simplicity, we assume that n is a power of 4, and that our subproblems are of integer size.

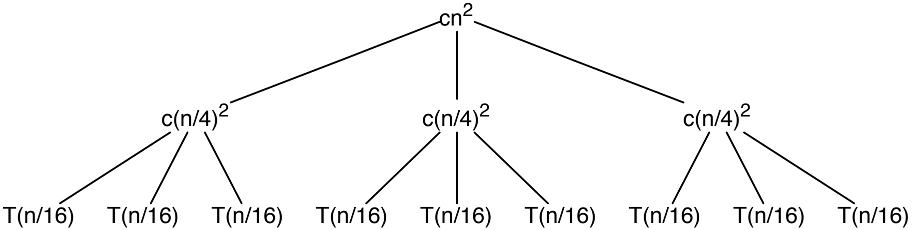
Expansion

$T(n)$

Expansion



Expansion

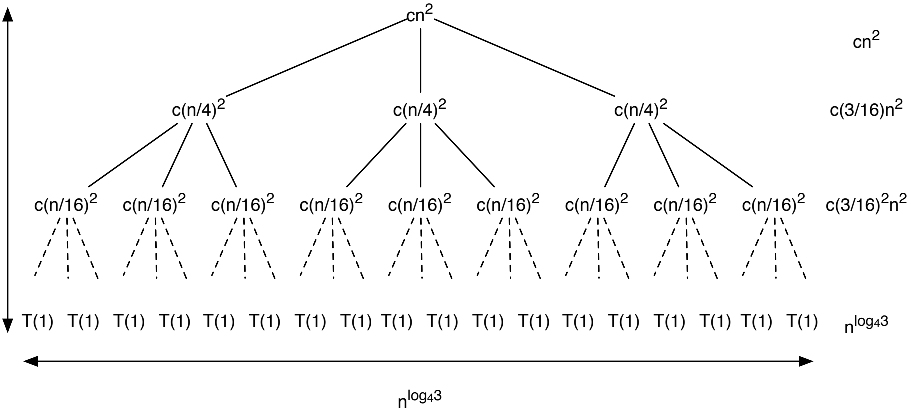


cn2
c(n/4)2
c(n/4)2
c(n/4)2
T(n/16)
T(n/16)
T(n/16)
T(n/16)
T(n/16)
T(n/16)
T(n/16)
T(n/16)
T(n/16)

Expansion

- At each expansion, our subproblem size decreases by a factor of 4.
- For a node of depth i, the subproblem size is $n/4^i$.
- We stop when $n = 1$ i.e. when $n/4^i = 1$ or equivalently, when $i = \log_4 n$.

Expansion



cn2
c(n/4)2
c(n/4)2
c(n/4)2
c(n/16)2
c(n/16)2
c(n/16)2
c(n/16)2
c(n/16)2
c(n/16)2
c(n/16)2
c(n/16)2
c(n/16)2
T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1)
cn2
c(3/16)n2
c(3/16)2n2
nlog43
nlog43

Expansion

$$
\begin{aligned}
T(n) &= cn^2 + 3/16cn^2 + (3/16)^2 cn^2 + \ldots + (3/16)^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n-1} (3/16)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} (3/16)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= 16/3cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2)
\end{aligned}
$$

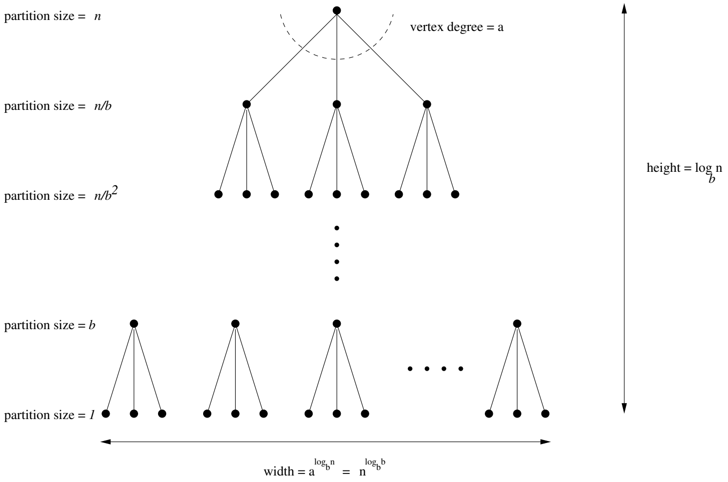
Now we verify

- We use this in the substitution method.
- Base case (n=1) holds as $1 \leq d$ for $d \geq 1$.

$$
\begin{aligned}
T(n) &\leq 3T(\lfloor n/4 \rfloor) + cn^2 \\
&\leq 3d\lfloor n/4 \rfloor^2 + cn^2 \\
&\leq 3d(n/4)^2 + cn^2 \\
&= (3/16)dn^2 + cn^2 \\
&\leq dn^2
\end{aligned}
$$

- Which holds when $d \geq (16/13)c$.

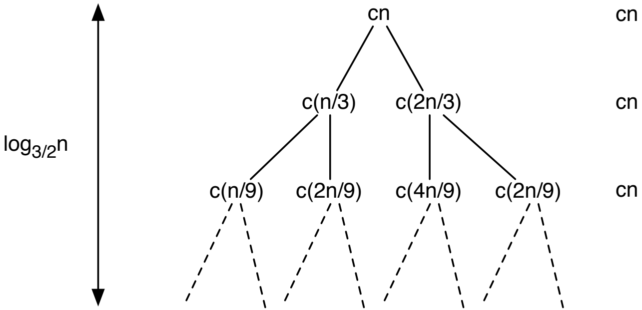
In General



Decomposition into a problems of size n/b

Another Example

$$T(n) = T(n/3) + T(2n/3) + O(n)$$



- Total cost: $O(n \log n)$
- This is a rough calculation based on the tree; we do not take the exact number of leaves into account!

The Master Method

- The master method provides a "cook book" approach to solving recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

- Where $a \geq 1$, $b > 1$ and $f(n)$ is an asymptotically positive function.

The Master Method

Let $a \geq 1$ and $b > 1$ be constants, $f(n)$ be a function, and $T(n)$ defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

Where n/b includes $\lfloor n/b \rfloor$ and $\lceil n/b \rceil$. Then T(n) has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

The Master Method, What does it mean?

Let $a \geq 1$ and $b > 1$ be constants, $f(n)$ be a function, and $T(n)$ defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

Where n/b includes $\lfloor n/b \rfloor$ and $\lceil n/b \rceil$. Then T(n) has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

- In all cases, we compare $f(n)$ with $n^{log_b a}$, and the larger of the two functions determines the solution to the recurrence.
- The intuition is that if the cost of combining dominates, that's the cost of the function, otherwise, the cost of computing sub solutions dominates.
- Note that these three cases do not cover all possibilities for $f(n)$.

Using the Master Method

$$T(n) = 9T(n/3) + n$$

- Here, $a = 9, b = 3, f(n) = n$
- Therefore, $n^{log_b a} = n^{log_3 9} = n^2$
- So $f(n) = n = n^1 = O(n^{2-1})$
- And thus, condition 1 holds.
- $T(n) = \Theta(n^2)$

Example 2

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

- $a = 1, b = 3/2, f(n) = 1$
- $n^{log_b a} = n^{log_{3/2} 1} = n^0 = 1$
- Case 2 applies as $f(n) = \Theta(n^{log_b a}) = \Theta(1)$
- Solution is thus $T(n) = \Theta(\log n)$

Example 3

$$T(n) = 3T(n/4) + n \log n$$

- $a = 3, b = 4, f(n) = n \log n$
- $n^{\log_b a} = n^{log_4 3} = O(n^{0.793})$
- $f(n) = \Omega(n^{log_4 3+\epsilon})$ for $\epsilon \approx 0.2$, case 3 applies if we can show the regularity condition.
- For large n, we have $af(n/b) = 3(n/4)\log(n/4) \leq (3/4)n \log n = cf(n)$ for $c = 3/4$.
- So the solution is $\Theta(n \log n)$

Example 4

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

- $a = 2, b = 2, f(n) = n \log n$
- $n^{\log_b a} = n$
- So we need to find ϵ such that $n \log n \geq kn^{log_2 2 + \epsilon} = kn^{\epsilon}$
- So we need, $n \log n/n = \log n \geq kn^{\epsilon}$
- But this does not hold for any positive ϵ or k for sufficiently large n.
- This case falls between case 2 and 3 of the master method.

Proof of the Master Method

- We've been using the master method as a black box, giving it some recurrence and getting a solution.
- But how do we know the master method is correct?
- We need to prove it.

Proof of the Master Method

- We've been using the master method as a black box, giving it some recurrence and getting a solution.
- But how do we know the master method is correct?
- We need to prove it.
- But we won't do that here, take a look at Cormen Chapter 4 for details.

Where are we?

- We now know how to compute the bounds of many different types of recurrences via formal proof, or by using the master method.
- More generally, we've dealt with many different aspects of algorithms, including correctness and complexity.
- We've examined the effects of different data structures on algorithms (particularly w.r.t space and time).
- We've examined sorting algorithms as exemplar algorithms on which to hone our skills.
- These skills are critical when designing an algorithm to solve a problem; they mean the difference between a program that takes hours/weeks to run (if not more), and one that takes seconds.
- In the remainder of this course, you'll be examining the properties of many other classes of algorithms, used in a variety of domains.