Time complexity analysis

Ben Liu

ben_0522@ku.edu

Definition

- Time complexity analysis is also called, running time (complexity) analysis. It refers to the analysis of the running time of an **algorithm** with respect to different **size** of input.
- Algorithm is a sequence of pre-defined statements for some purpose, the running time of the algorithm depends on:
    1. The environment (language, hardware, ...)
    2. The implementation (bubble sort vs merge sort)
    3. The input size (sort 10 items vs 1000 items)

RAM Model

- RAM is an ideal model used to analyze the running time of an algorithm without taking into consideration the power of environment:
    1. Each basic operation takes constant time;
    2. Exactly one statement can be executed at one time;
    3. Any datum can be stored in one memory block;
    4. Any stored datum can be accessed at constant time;

Type of analysis

- Exact analysis
- Approximation analysis
- Asymptotic analysis

Example 1 – exact analysis

```
4   int a = 5;
5   int d = 2;
6   int sum = 0;
7   int k = 100;
8   for(int i = 0; i <= k; ++i)
9   {
10    int a_prime = a + i*d;
11    sum = sum + a_prime;
12  }
```

- Operations:
  1. Assign, (line 4, line 5, ...) takes t_1;
  2. Compare, (line 8), takes t_2;
  3. Addition, (line 8, line 10) takes t_3;
  4. Multiplication, (line 10) takes t_4;
- Counting operations:
  1. Line 4 to line 7 were executed once;
  2. for-loop was executed 101 times;
  3. Note only initialize once.

Example 1 – exact analysis

```
int a = 5;
int d = 2;
int sum = 0;
int k = 100;
for(int i = 0; i <= k; ++i)
{
    int a_prime = a + i*d;
    sum = sum + a_prime;
}
```

- So, final counts of operations
  1. 308 assign;
  2. 102 compare (think of why?);
  3. 303 addition;
  4. 101 multiplication;
- All together:

$$T = 308t_1 + 102t_2 + 303t_3 + 101t_4$$

Example 1 – approximation analysis

```
4   int a = 5;
5   int d = 2;
6   int sum = 0;
7   int k = 100;
8   for(int i = 0; i <= k; ++i)
9   {
10    int a_prime = a + i*d;
11    sum = sum + a_prime;
12  }
```

- Operation blocks:
  1. Initialization , line 4, to line 7;
  2. for-loop, line 8 to line 12;
- Note that the running time of for-loop dominates Initialization. So it’s reasonable for us to use the running time of for-loop to approximate the running time of the algorithm.

Example 1 – approximation analysis

```
int a = 5;
int d = 2;
int sum = 0;
int k = 100;
for(int i = 0; i <= k; ++i)
{
  int a_prime = a + i*d;
  sum = sum + a_prime;
}
```

- Now:

$$T = 304t_1 + 102t_2 + 303t_3 + 101t_4$$

- Before:

$$T = 308t_1 + 102t_2 + 303t_3 + 101t_4$$

- Not much difference, but it will be much easier for us, especially when the algorithm has tons of operation blocks.

Asymptotic analysis

- Asymptotic analysis is more useful when the input size changes, but here in example 1, the algorithm in example 1 doesn’t accept input, or we can say the input size is constant.
- Let’s see the definition of asymptotic analysis.

Asymptotic analysis

- In mathematic, asymptotic analysis studies the behavior of function $f(n)$ when n is very large.
- There are different notations of asymptotic, the following 3 are most used:
  1. Big-O, we focus on Big-O through this tutorial;
  2. Little-o;
  3. Big-Ω;
  4. Little-ω;
  5. Big-Θ;

Asymptotic analysis

- Definition of Big-O:

  for any given function $f(n)$, if there exists two positive constants k, n_0 and another function $g(n)$, we say that $f(n) \in O(g(n))$ if and only if $f(n) \leq k \cdot g(n)$ when $n \geq n_0$. (Upper bound)

- Note that upper bound doesn't have to be tight, for example:

  given $f(n) = n$, $g(n) = n^2$, $k = 1$, $n_0 = 1$,

  apparently for $n \geq n_0$, $f(n) \leq k \cdot g(n)$, or for $n \geq 1$, $n \leq 1 \cdot n^2$, so $n \in O(n^2)$.

- But,

  if $g(n) = n^3$, we still have for $n \geq 1$, $n \leq 1 \cdot n^3$, so $n \in O(n^3)$ as well.

Asymptotic analysis

- Definition of Little-o:

  for any given function $f(n)$, if there exists two positive constants k, n_0 and another function $g(n)$, we say that $f(n) \in o(g(n))$ if and only if $f(n) < k \cdot g(n)$ when $n \geq n_0$. (Tight upper bound)

- Definition of Big-Ω:

  for any given function $f(n)$, if there exists two constants k, n_0 and another function $g(n)$, we say that $f(n) \in \Omega(g(n))$ if and only if $f(n) \geq k \cdot g(n)$ when $n \geq n_0$. (Lower bound)

Asymptotic analysis

- Definition of Little-ω:

  for any given function $f(n)$, if there exists two positive constants k, n_0 and another function $g(n)$, we say that $f(n) \in \omega(g(n))$ if and only if $f(n) > k \cdot g(n)$ when $n \geq n_0$. (Tight lower bound)

- Definition of Big-Θ:

  for any given function $f(n)$, if there exists three constants k_1, k_2, n_0 and another function $g(n)$, we say that $f(n) \in \Theta(g(n))$ if and only if $k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$ when $n \geq n_0$. (Tight bound)

- Let' move on another example, sequential search.

Example 2

```cpp
// A is an array of N integer items,
// k is the value we are looking for.
bool sequentialSearch(cont std::vector<int> & A, int k)
{
  if(A.empty())
    return false;
  for(int i = 0; i < A.size(); ++i)
  {
    if(A[i] == k) return true;
  }
  return false;
}
```

- Operations:
  1. empty check, t_1;
  2. return, t_2;
  3. assign, t_3;
  4. compare <, t_4;
  5. get size, t_5;
  6. addition, t_6;
  7. compare ==, t_7;
- Dominant statement, for-loop.

Example 2

```cpp
// A is an array of N integer items,
// k is the value we are looking for.
bool sequentialSearch(cont std::vector<int> & A, int k)
{
  if(A.empty())
    return false;
  for(int i = 0; i < A.size(); ++i)
  {
    if(A[i] == k) return true;
  }
  return false;
}
```

- Trivial case, if A is empty, the for-loop will be executed 0 time.
- Best case, if the first item of A is the item we are looking for, the for-loop will be executed 1 time.
- Worst case, if the last item of A is the item we are looking for, the for-loop will be executed N times.

Example 2

```cpp
// A is an array of N integer items,
// k is the value we are looking for.
bool sequentialSearch(cont std::vector<int> & A, int k)
{
  if(A.empty())
    return false;
  for(int i = 0; i < A.size(); ++i)
  {
    if(A[i] == k) return true;
  }
  return false;
}
```

- Average case: assuming each item has equal probability p to be the item we are looking for, the for-loop is expected to be executed Np times.
- Which case should we use?

Example 2 - approximation analysis

```cpp
// A is an array of N integer items,
// k is the value we are looking for.
bool sequentialSearch(const std::vector<int> & A, int k)
{
  if(A.empty())
    return false;
  for(int i = 0; i < A.size(); ++i)
  {
    if(A[i] == k) return true;
  }
  return false;
}
```

- In the worst case:

$$T = (t_3 + t_4 + t_5 + t_6) \times N + t_3 + t_7$$

Example 2 - approximation analysis

$$T = (t_3 + t_4 + t_5 + t_6) \times N + t_3 + t_7$$

Let's assume:

$a = (t_3 + t_4 + t_5 + t_6)$

b$= t_3 + t_7$

Then,

$$T = aN + b$$

This is a function of N, so,

$$T(N) = aN + b$$

Example 2 - asymptotic analysis

- Given $T(N) = aN + b$, let's prove $T(N) \in O(N)$.

  Let $g(N) = N$, $k = a + b$, $n_0 = 1$,

  Obviously, for $N \geq 1$, which is $N \geq n_0$,

$$b \leq bN$$
$$aN + b \leq aN + bN$$
$$T(N) \leq aN + bN$$
$$T(N) \leq (a + b)N$$
$$T(N) \leq (a + b)g(N)$$
$$T(N) \leq kg(N)$$

  So, we proved that for $N \geq 1$, $T(N) \in O(N)$.

Example 3

```cpp
// A is an array of N integer items, sorted in non-descending order.
// k is the value we are looking for.
bool binarySeach(cont std::vector<int> & A, int k)
{
  if(A.empty())
    return false;
  int l = 0, h = A.size()-1;
  int m = 0;
  while(l < h)
  {
    m = floor( ( l + h ) / 2 );
    if(A[m] == k)       return true;
    else if(A[m] < k)  l = m + 1;
    else               h = m - 1;
  }
  return false;
}
```

- The while-loop dominates the algorithm.
- For simplicity, let's assume the while-loop takes a time to be executed once.
- In the worst case, the item we are looking for is the first or the last item of the array, the while-loop will be executed $\log_2(N)$ times.

Example 3

- Given $T(n) = a\log_2(n)$, let's prove $T(n) \in O\left(\log_2(n)\right)$.

  Let $g(n) = \log_2(n)$, $k = 2a$, $n_0 = 1$,

  obviously, $a\log_2(n) \leq 2a\log_2(n)$ when $n > 1$.

  Hence, $T(n) \in O\left(\log_2(n)\right)$.

Example 4

```cpp
// A is an array of N integer items to be sorted
void bubbleSortNonDesending(std::vector<int> & A)
{
  bool ff_not_sorted = true;
  while(ff_not_sorted)
  {
    ff_not_sorted = false;
    for(int i = 1; i < A.size(); ++i)
    {
      if(A[i-1] > A[i])
      {
        int temp = A[i-1];
        A[i-1] = A[i];
        A[i] = temp;
        ff_not_sorted = true;
      }
    }
  }
}
```

- The while-loop dominates the algorithm. The for-loop dominates the body of while-loop.
- Inside the for-loop, assume it takes a time to execute the if-statement if the condition is true, otherwise it takes b time to execute the if-statement.

Example 4 – an instance of worst case

```cpp
// A is an array of N integer items to be sorted
void bubbleSortNonDesending(std::vector<int> & A)
{
  bool ff_not_sorted = true;
  while(ff_not_sorted)
  {
    ff_not_sorted = false;
    for(int i = 1; i < A.size(); ++i)
    {
      if(A[i-1] > A[i])
      {
        int temp = A[i-1];
        A[i-1] = A[i];
        A[i] = temp;
        ff_not_sorted = true;
      }
    }
  }
}
```

- In the worst case, the input array has N items in ascending order.
- For instance,

$$A = \{5, \quad 4, \quad 3, \quad 2, \quad 1\}$$

Example 4 – an instance of worst case

```cpp
// A is an array of N integer items to be sorted
void bubbleSortNonDesending(std::vector<int> & A)
{
  bool ff_not_sorted = true;
  while(ff_not_sorted)
  {
    ff_not_sorted = false;
    for(int i = 1; i < A.size(); ++i)
    {
      if(A[i-1] > A[i])
      {
        int temp = A[i-1];
        A[i-1] = A[i];
        A[i] = temp;
        ff_not_sorted = true;
      }
    }
  }
}
```

- The $1st$ repeat of while-loop:
- Before for-loop:

$$A = \{5, \quad 4, \quad 3, \quad 2, \quad 1\}$$

i = 1:

$$A = \{4, \quad 5, \quad 3, \quad 2, \quad 1\}$$

i = 2:

$$A = \{4, \quad 3, \quad 5, \quad 2, \quad 1\}$$

i = 3:

$$A = \{4, \quad 3, \quad 2, \quad 5, \quad 1\}$$

i = 4:

$$A = \{4, \quad 3, \quad 2, \quad 1, \quad 5\}$$

Example 4 – an instance of worst case

- The $2nd$ repeat of while-loop:
- Before for-loop:

$$A = \{4, \quad 3, \quad 2, \quad 1, \quad 5\}$$

i = 1:

$$A = \{3, \quad 4, \quad 2, \quad 1, \quad 5\}$$

i = 2:

$$A = \{3, \quad 2, \quad 4, \quad 1, \quad 5\}$$

i = 3:

$$A = \{3, \quad 2, \quad 1, \quad 4, \quad 5\}$$

i = 4:

$$A = \{3, \quad 2, \quad 1, \quad 4, \quad 5\}$$

- The $3rd$ repeat of while-loop:
- Before for-loop:

$$A = \{3, \quad 2, \quad 1, \quad 4, \quad 5\}$$

i = 1:

$$A = \{2, \quad 3, \quad 1, \quad 4, \quad 5\}$$

i = 2:

$$A = \{2, \quad 1, \quad 3, \quad 4, \quad 5\}$$

i = 3:

$$A = \{2, \quad 1, \quad 3, \quad 4, \quad 5\}$$

i = 4:

$$A = \{2, \quad 1, \quad 3, \quad 4, \quad 5\}$$

Example 4 – an instance of worst case

- The 4th repeat of while-loop:
- Before for-loop:

$A = \{2, \quad 1, \quad 3, \quad 4, \quad 5\}$

i = 1:

$A = \{1, \quad 2, \quad 3, \quad 4, \quad 5\}$

i = 2:

$A = \{1, \quad 2, \quad 3, \quad 4, \quad 5\}$

i = 3:

$A = \{1, \quad 2, \quad 3, \quad 4, \quad 5\}$

i = 4:

$A = \{1, \quad 2, \quad 3, \quad 4, \quad 5\}$

- The 5th repeat of while-loop:
- Before for-loop:

$A = \{1, \quad 2, \quad 3, \quad 4, \quad 5\}$

i = 1:

$A = \{1, \quad 2, \quad 3, \quad 4, \quad 5\}$

i = 2:

$A = \{1, \quad 2, \quad 3, \quad 4, \quad 5\}$

i = 3:

$A = \{1, \quad 2, \quad 3, \quad 4, \quad 5\}$

i = 4:

$A = \{1, \quad 2, \quad 3, \quad 4, \quad 5\}$

Example 4 – an instance of worst case

```cpp
// A is an array of N integer items to be sorted
void bubbleSortNonDesending(std::vector<int> & A)
{
  bool ff_not_sorted = true;
  while(ff_not_sorted)
  {
    ff_not_sorted = false;
    for(int i = 1; i < A.size(); ++i)
    {
      if(A[i-1] > A[i])
      {
        int temp = A[i-1];
        A[i-1] = A[i];
        A[i] = temp;
        ff_not_sorted = true;
      }
    }
  }
}
```

- Since the 5th repeat of while-loop doesn't swap any item, the flag ff_not_sorted will be true, so the while-loop condition will be false, the algorithm will be terminated.

Example 4 – an instance of worst case

```cpp
// A is an array of N integer items to be sorted
void bubbleSortNonDesending(std::vector<int> & A)
{
  bool ff_not_sorted = true;
  while(ff_not_sorted)
  {
    ff_not_sorted = false;
    for(int i = 1; i < A.size(); ++i)
    {
      if(A[i-1] > A[i])
      {
        int temp = A[i-1];
        A[i-1] = A[i];
        A[i] = temp;
        ff_not_sorted = true;
      }
    }
  }
}
```

- In summary,

$$T(5) = (4a) + (3a + b) + (2a + 2b) + (a + 3b) + (4b) = (a + b)\frac{4(1 + 4)}{2}$$

- More generally,

$$T(n) = (a + b)\frac{n(n-1)}{2}$$

Example 4

- Given $T(n) = (a+b)\frac{n(n-1)}{2}$, let's prove $T(n) \in O(n^2)$.

Let $g(n) = n^2$, $k = a + b$, $n_0 = 1$,

obviously, when $n > 1$,

$$n \le n^2,$$

$$\frac{(a+b)}{2} n \le \frac{(a+b)}{2} n^2$$

$$\frac{(a+b)}{2} n^2 + \frac{(a+b)}{2} n \le \frac{(a+b)}{2} n^2 + \frac{(a+b)}{2} n^2$$

$$T(n) = \frac{(a+b)}{2} n^2 - \frac{(a+b)}{2} n < \frac{(a+b)}{2} n^2 + \frac{(a+b)}{2} n \le (a+b)n^2$$

Hence, $T(n) \in O(n^2)$.

Example 5 – Multiplication of matrices

```
1  import numpy as np
2
3  # A and B are both n-by-n matrices
4  def multipleMatrices(A, B):
5      if(A.shape[1] != B.shape[0]):
6          return None
7      ans = np.zeros((A.shape[0], B.shape[1]))
8      for i in range(A.shape[0]):
9          for j in range(B.shape[1]):
10             for k in range(A.shape[1]):
11                 ans[i,j] += A[i,k]*B[k,j]
12     return ans
```

- The nested for-loop dominates the algorithm.
- For simplicity, assuming both matrices are n-by-n square matrices.
- Assume it takes a time to execute line 11.

- In summary,

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} a = an^3$$

Example 6 – merge sort

```cpp
void mergeSort(vector<int>& A)
{
  mergeSortSplit(A, 0, A.size());
}

void mergeSortSplit(vector<int>& A, int begin, int end)
{
  if(end - begin <= 1) return;   //one item itself is sorted
  int middle = int((begin + end) / 2);
  mergeSortSplit(A, begin,  middle);
  mergeSortSplit(A, middle, end);
  mergeSortMerge(A, begin, middle, end);
}

void mergeSortMerge(vector<int>& A, int begin, int middle, int end)
{
    vector<int> B(A.begin()+begin, A.begin()+middle);
    vector<int> C(A.begin()+middle, A.begin()+end);

    int p = 0, q = 0, k = begin;
    while(p < B.size() && q < C.size())
    {
        if(B[p] < C[q])  A[k++] = B[p++];
        else             A[k++] = C[q++];
    }
    while(p < B.size())  A[k++] = B[p++];
    while(q < C.size())  A[k++] = C[q++];
}
```

- Merge sort is a typical divide and conquer algorithm.
- During the divide stage, the original problem is divided into equally two half, so

$$T(N) = 2T\left(\frac{N}{2}\right) + f(N)$$

- $f(N)$ is the time taken to merge the subproblem (function $mergeSortMerge$).

Example 6 – merge sort

```
void mergeSort(vector<int>& A)
{
  mergeSortSplit(A, 0, A.size());
}

void mergeSortSplit(vector<int>& A, int begin, int end)
{
  if(end - begin <= 1) return;   //one item itself is sorted
  int middle = int((begin + end) / 2);
  mergeSortSplit(A, begin,  middle);
  mergeSortSplit(A, middle, end);
  mergeSortMerge(A, begin, middle, end);
}

void mergeSortMerge(vector<int>& A, int begin, int middle, int end)
{
    vector<int> B(A.begin()+begin, A.begin()+middle);
    vector<int> C(A.begin()+middle, A.begin()+end);

    int p = 0, q = 0, k = begin;
    while(p < B.size() && q < C.size())
    {
        if(B[p] < C[q])  A[k++] = B[p++];
        else             A[k++] = C[q++];
    }
    while(p < B.size())  A[k++] = B[p++];
    while(q < C.size())  A[k++] = C[q++];
}
```

- In function *mergeSortMerge*, the array B and array C sorted the sorted sub-array and the 3 while-loop will merge the 2 sorted subarray into 1 array.
- In each repeat of any of the while-loop, one element is put into where it's supposed to be.

Example 6 – merge sort

```cpp
void mergeSort(vector<int>& A)
{
  mergeSortSplit(A, 0, A.size());
}

void mergeSortSplit(vector<int>& A, int begin, int end)
{
  if(end - begin <= 1) return;   //one item itself is sorted
  int middle = int((begin + end) / 2);
  mergeSortSplit(A, begin,  middle);
  mergeSortSplit(A, middle, end);
  mergeSortMerge(A, begin, middle, end);
}

void mergeSortMerge(vector<int>& A, int begin, int middle, int end)
{
    vector<int> B(A.begin()+begin, A.begin()+middle);
    vector<int> C(A.begin()+middle, A.begin()+end);

    int p = 0, q = 0, k = begin;
    while(p < B.size() && q < C.size())
    {
        if(B[p] < C[q])  A[k++] = B[p++];
        else             A[k++] = C[q++];
    }
    while(p < B.size())  A[k++] = B[p++];
    while(q < C.size())  A[k++] = C[q++];
}
```

- The 3 while-loops are dominant. Assume it takes takes a time to execute one repeat of the while-loop.
  $$f(N) = aN$$
- Where, N is the total sizes of the 2 subproblems. So,
  $$T(N) = 2T\left(\frac{N}{2}\right) + aN$$
- Solve it,
  $$T(N) = N + \mathrm{a} \cdot \mathrm{N} \cdot \log_2(N)$$
  $$T(N) \in \mathrm{O}\big(\mathrm{N}\log_2(N)\big)$$

Example 6 – merge sort

$T(N) = 2T\left(\frac{N}{2}\right) + aN$, so, $T\left(\frac{N}{2}\right) = 2T\left(\frac{N}{2^2}\right) + a\frac{N}{2}$, $T\left(\frac{N}{2^2}\right) = 2T\left(\frac{N}{2^3}\right) + a\frac{N}{2^2}$

Plugging in, we have

$T(N) = 2\left(2T\left(\frac{N}{2^2}\right) + a\frac{N}{2}\right) + aN = 2^2T\left(\frac{N}{2^2}\right) + aN + aN$

$T(N) = 2^2\left(2T\left(\frac{N}{2^3}\right) + a\frac{N}{2^2}\right) + aN + aN = 2^3T\left(\frac{N}{2^3}\right) + aN + aN + aN$

..., assuming $N = 2^k$

$T(N) = 2^kT\left(\frac{N}{2^k}\right) + akN = N + a\mathrm{N}\log_2(N)$

Example 7 – compute Fibonacci 1

```
// naive computation of fibonacci
void fibonacci(int N)
{
  int ans = 0
  if(N == 0)       return 0;
  else if(N < 2)   return 1;
  int f1 = 0, f2 = 1;
  for(int i = 2; i <= N; ++i)
  {
    int ans = f1 + f2;
    f1 = f2;
    f2 = ans;
  }
  return ans;
}
```

- The for-loop dominate the algorithm.
- Assume it takes a time to execute the body of for-loop.
- $T(N) = a(N-1) \in O(N)$.

Example 8 – compute Fibonacci 2

```
// naive computation of fibonacci
void fibonacci(int N)
{
  if(N == 0)  return 0;
  if(N == 1)  return 1;
  return fibonacci(N-1)+fibonacci(N-2);
}
```

- The running time of the algorithm:

  $$T(N) = T(N-1) + T(N-2) + 1$$

- Solving the linear recursive equation:

  $$T(N) = \left(\frac{1+\sqrt{5}}{2}\right)^N + \left(\frac{1-\sqrt{5}}{2}\right)^N$$

Example 8

- Given $T(n) = \left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n$, let's prove $T(n) \in O(2^n)$.

  Let $g(n) = 2^n$, $k = 2$, $n_0 = 1$,

  obviously, when $n > 1$,

$$\frac{1+\sqrt{5}}{2} \leq 2 \rightarrow \left(\frac{1+\sqrt{5}}{2}\right)^n \leq 2^n$$

$$\frac{1-\sqrt{5}}{2} \leq 2 \rightarrow \left(\frac{1-\sqrt{5}}{2}\right)^n \leq 2^n$$

$$T(n) \leq 2 \cdot 2^n$$

  Hence, $T(n) \in O(2^n)$.