Contents

1 Runtime analysis of algorithm

Definition of Runtime

We use atomic operations instead of real running time of the algorithm since different devices have different real running time but must have the same atomic operations.
Given a set A of atomic operations and an algorithm A,

$$\begin{aligned} T(A, I) &= \text{runtime of A on input I} \\ &= \#\text{ atomic operations} \end{aligned}$$

Assumptions:

1. $T(A, I)$ only depends on the input I,
   counter-example: coin-flipping doesn't fulfill this assumption.
2. all the atomic operations are equal, however, some of them may take more time in the real world.
3. $T(A, I)$ only depends on the size of input I, denoted as $|I|$, but not a particular input with that size.

Worse Case Execution Time (WCET)

$$T(A, n) = \max_{|I|=n} T(A, I)$$

Asymptotic Analysis

- Two algorithm with runtimes $f(n)$, and $g(n)$
  $5n + 5 = O(n)$, $100n = O(n)$, they are asymptotic equivalent.
- Formal definition:
  We say $f \in O(g)$ if
  $$\exists c > 0, \forall n \geq n_0, f(n) \leq cg(n)$$
- Definition 2:
  we say $f \in \Omega(g)$ if
  $$\exists n_0, \exists c > 0, \forall n > n_0, cf(n) \geq g(n)$$
- Definition 3:
  $$f \in \Theta(g) \iff f \in O(g) \wedge f \in g \in O(f)$$
- Definition 4:
  $$f \in o(g) \iff f \in O(g) \wedge f \in g \notin O(f)$$
  $$f \in \omega(g) \iff f \in \Omega(g) \wedge f \in g \notin \Omega(f)$$

Fibonacci Numbers

$$f_0 = 1, f_1 = 1, f_i = f_{i-1} + f_{i-2}$$

```
int fibo(const int& n){
   if (n==0 || n==1) return 1;
   return fibo(n-1) + fibo(n-2);
}
```

$$T(A,n) = \begin{cases} O(1), & \text{if } n = 0, \text{or } n = 1 \\ T(A,n-1) + T(n-2) + O(1) & \text{otherwise} \end{cases}$$
$$\implies T(n) > f_n$$

That is, since f_n increase exponentially, the runtime of the algorithm is even worse than the exponential runtime, *which is not desirable*. Instead of using recursion, we can simply use array

```
int fibo(const int& n){
    int* A = new int[n+1];
    A[0] = A[1] =1;
    for (int i=2; i<n+1; i++){
        A[i] = A[i-1] + A[i-2];
    }
    int result = A[n];
    delete [] A;
    return result;
}
```

$$T(A,n) \in O(n)$$

We can see that the latter the algorithm is better since it's faster!

Maximum Contiguous Subsequence
Input: an array A of integers, e.g., $\{10, 5, -2, -3, 10, 12, 0\}$
Output: two indices i, j, such that $\sum_{k=i}^{j} A[k]$ is maximized.
There are many algorithms to solve this problem:

1. By brute force, trying out all the possibilities

```
const int MAXN = 1e6;
int A[MAXN];

int main(){
   int n;
   cin>> n;
   for (int i=0; i<=n; i++){
      cin>> A[i];
   }
   int ans = 0;
   for (int i=0; i<=n; i++){
      int sum = 0;
      for (int j=i; j<=n; i++){
         sum += A[j];
         if (sum > ans){
            ans = sum;
         }
      }
   }
   cout<< ans;
   return 0;
}
```

$$T(A,n) \in O(n^2)$$

2. Divide and conquer